Universidad Politécnica de Madrid
Facultad de Informática

Trabajo de Fin de Grado

Diseño de una arquitectura escalable y de alta disponibilidad para un sistema middleware

Autor:
Fernando Godino González

Tutora:
Genoveva López Gómez

Junio 2013

AGRADECIMIENTOS

Quiero aprovechar estas líneas para dar las gracias a todas las personas que me han apoyado a lo largo de estos meses de duro trabajo y en general en toda la carrera.

A mi familia que siempre ha apoyado en mis decisiones y me ha dado buenos consejos. A mis padres y a mis hermanas por aguantarme en los momentos de estrés y darme ánimos para seguir adelante.

A mi novia Bego que ha sabido entender tantas horas dedicadas a la carrera, y que me ha obligado a salir de casa cuando la inspiración no acudía. Gracias por sus buenas críticas a todos mis trabajos.

A mis amigos de la Facultad, en especial a mis compañeros del Laboratorio de Innovación abierta: Aitor, David, CD, Sonia, Alberto, Salva y Obaib. Gracias por los buenos momentos en el laboratorio y por las charlas filosóficas. Ha sido un placer trabajar con vosotros.

Finalmente, también quiero dar las gracias a todos los profesores de la Facultad de Informática UPM, ya que gracias a ellos he conocido el mundo fascinante que es la informática. Agradecimientos especiales a mis tutores por ayudarme y orientarme a lo largo de este Proyecto.

Muchas Gracias.

Índice general

Índice de figuras

Índice de algoritmos

Resumen

El objetivo de este proyecto es el estudio de soluciones de escalabilidad y alta disponibilidad en sistemas distribuidos, así como su implantación en aquel de los sistemas analizados por Telefónica Digital, PopBox y Rush, que se consideré más adecuado.

Actualmente, muchos servicios y aplicaciones están alojados directamente en la Web, permitiendo abaratar el uso de ciertos servicios y mejorando la productividad y la competitividad de las empresas que los usan. Este crecimiento de las tecnologías en cloud experimentado en los últimos años plantea la necesidad de realizar sistemas que sean escalables, fiables y estén disponibles la mayor parte del tiempo posible. Un fallo en el servicio no afecta a una sola empresa, sino a todas las que están haciendo uso de dicho servicio.

A lo largo de este proyecto se estudiarán las soluciones de alta disponibilidad y escalabilidad implementadas en varios sistemas distribuidos y se realizará una evaluación crítica de cada una de ellas. También se analizará la idoneidad de estas soluciones para los sistemas en los que posteriormente se aplicarán: PopBox y Rush.

Se han diseñado diferentes soluciones para las plataformas implicadas, siguiendo varias aproximaciones y realizando un análisis exhaustivo de cada una de ellas, teniendo en cuenta el rendimiento y fiabilidad de cada aproximación. Una vez se ha determinado cuál es la estrategia más adecuada, se ha realizado una implementación fiable del sistema. Para cada uno de los módulos implementados se ha llevado a cabo una fase de testing unitario y de integración para asegurar el buen comportamiento del sistema y la integridad de éste cuando se realicen cambios.

Específicamente, los objetivos que se alcanzarán son los siguientes:

1. Análisis exhaustivo de los sistemas de escalabilidad y alta escalabilidad que existen actualmente.

2. Diseño de una solución general HA[1] y escalable teniendo en cuenta el objetivo anterior.

3. Análisis de la idoneidad de los sistemas PopBox y Rush para el diseño de un entorno distribuido escalable.

4. Diseño e implantación de una solución *ad-hoc* en el sistema elegido.

Abstract

The aim of this project is the study of solutions in scalability and high availability in distributed systems, and also its implementation in one of the systems developed by Telefónica I+D, PopBox and Rush, deemed more suitable.

[1] High Availability

Nowadays, a lot of services and applications are stored directly in the Web, allowing companies to reduce the costs of using certain services and improving the productivity and competitiveness of those who use these services. This increase of the use of cloud tecnologies experimented in the last few years has led to the need of developing high available, scalable, and reliable systems. A failure in the service does not affect a single company but all the companies using this service.

Throughout this project, I will study several solutions in High Availability and Scalability developed in some distributed systems and I will make a critic analysis of each one. Also I will analize the suitability of these solutions in the systems in which they will be applied: PopBox and Rush.

I have designed different solutions for the platforms involved, following several approaches and making an exhaustive analysis of each one, taking into account their performance and reliability of each approach. Once I had determined which is the best strategy, I have developed a reliable implementation of the system. For each module implemented, I have carried out a set of unitary and integration tests to ensure the good behaviour of the system and the integrity of it when it changes.

Specifically, the objectives to be achieved are as follows:

1. Exhaustive analysis of the systems in scalability and high availability that currently exist.
2. Design of a general solution taking into account the previous point.
3. Analysis of the suitability of the sistems PopBox and Rush for the design of a scalable distributed system.
4. Design and implementation of an *ad-hoc* solution in the chosen system.

Capítulo 1

Introducción

1.1. Introducción

La forma en que las empresas ofrecen servicios TI a sus usuarios está sufriendo un profundo cambio en los últimos años. El software ya no se concibe como un producto por el que se paga una vez, se adquiere una licencia y se comienza a usar. El mercado de las TI evoluciona hacia un modelo de negocio de "pago por uso", es decir, el usuario final alquila un servicio y paga proporcionalmente a la intensidad de uso de este. Este modelo se conoce como SaaS[1], y basa su éxito en la economía de escala; empresas del tamaño de BBVA han externalizado los servicios de correo electrónico y de colaboración (Calendar, Docs, Groups, Sites y vídeos) mediante Google, ahorrando costes de entre un 50 % y un 70 % en infraestructura y mantenimiento[Men12].

Según Gartner, una de las empresas analistas de tecnologías de la información más importantes del mundo, la inversión total en SaaS por parte de las empresas se incrementará un 17.9 % en 2013, alcanzando un valor de 14.5 billones de dólares[Kan12] (11000 millones de euros). Este crecimiento viene dado en parte por el aumento de soluciones PaaS (Platform as a Service) y por la familiarización de las empresas con este modelo.

El Saas es un modelo de distribución de software que permite a los usuarios alquilarlo desde cualquier dispositivo, a través de la red. Además, el usuario no pagará por disponer de cierto software, sino que pagará por hacer uso de éste (as a Service). Este nuevo paradigma permite a cualquier empresa o usuario utilizar recursos software sin necesidad de realizar grandes inversiones iniciales, por lo tanto el periodo de recuperación de la inversión es nulo. Este modelo unifica el concepto de producto y el de servicio para dotar a las empresas de una solución que permita optimizar los recursos y gestionarlos de una forma flexible y adaptativa.

Las ventajas que ofrece SaaS para las empresas son una reducción en la inversión inicial en infraestructuras además de un ahorro considerable en mantenimiento de

[1]Software as a Service

máquinas y software. Adicionalmente, el soporte y las actualizaciones de software es más ágil, obteniendo un servicio continuamente actualizado. SaaS reduce el riesgo estratégico en TI, ya que no es necesario adquirir nuevo software ni pagar licencias. Esto permite a las empresas centrar sus esfuerzos en el negocio, obteniendo mejores resultados.

El frenético ritmo de desarrollo de las TI requiere de una amortización acelerada de los activos tecnológicos. Por tanto la utilización de SaaS, supone, desde el punto de vista económico, menores amortizaciones, aumento del activo disponible, menor necesidad de endeudamiento, además de permitir la rápida actualización tecnológica a bajo coste.

El concepto de SaaS está estrechamente relacionado con el Cloud Computing. La prestación del servicio se realiza mediante la red (normalmente Internet) y suele estar distribuido. Los primeros proveedores de servicios "cloud" a gran escala han sigo Google, Amazon y Microsoft, al estar dotados inicialmente de una infraestructura muy amplia y muy potente.

El amplio crecimiento de este modelo de negocio y las ventajas que este ofrece han llevado al departamento de Desarrollo Transversal de Telefónica I+D a convertir dos productos concebidos inicialmente como componentes (PopBox y Rush) en servicios.

Los servicios en la nube deberán soportar un crecimiento muy amplio y muy rápido, ya que es imposible prever la intensidad de uso que una empresa puede requerir de estos en un determinado momento. Un buen SaaS deberá ser capaz de aumentar o disminuir sus recursos para satisfacer las necesidades del cliente. Para que un sistema sea capaz de aumentar sus recursos sin disminuir su rendimiento debe ser escalable. Así mismo, el sistema debe responder rápidamente a cambios en la demanda por parte del usuario, es decir, debe ser elástico.

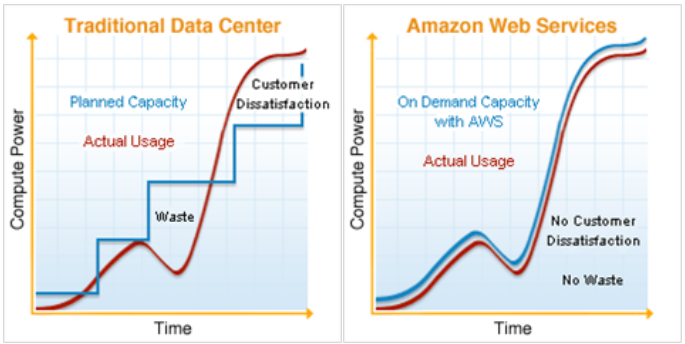

Figura 1.1.1: Diferencia entre un sistema poco elástico (izquierda) y un sistema con mucha elasticidad (derecha).

Las consecuencias que conllevan un fallo en un sistema Cloud no son las mismas que en un sistema tradicional *in-house*. La perdida de servicio o de datos en un sistema Cloud afecta a todos sus cliente. Por esta razón, los sistemas de Cloud Computing deben estar disponibles el mayor tiempo posible y proveer soluciones

para la recuperación de datos en caso de perdidas. En definitiva, un sistema cloud debe estar dotado de soluciones de Alta Disponibilidad.

El Laboratorio de Innovación Abierta UPM-Telefónica Digital está trabajando en dos sistemas SaaS, PopBox y Rush. Estos sistemas deben cumplir requisitos como fiabilidad, escalabilidad y alta disponibilidad para poder ser considerados como buenos sistemas SaaS.

Este proyecto consistirá en el análisis y el diseño de soluciones de Alta Disponibilidad, Escalabilidad[2] y Elasticidad[3] en uno de estos dos sistemas, concretamente PopBox, teniendo en cuenta las soluciones que actualmente existen en este campo. Al finalizar el desarrollo de este proyecto, el sistema PopBox podrá satisfacer la demanda de recursos de los usuarios de una manera dinámica y elástica, además de proveer soluciones de alta disponibilidad y seguridad de datos.

Los sistemas PopBox y Rush están basados en NodeJs como entorno de servidor, y en Redis como base de datos. Las soluciones de escalabilidad y alta disponibilidad, por lo tanto, deberán estar implementadas sobre estas dos tecnologías, algo nada trivial.

1.2. Objetivos

El presente trabajo se centrará en el diseño y desarrollo de una solución de alta disponibilidad, escalabilidad y elasticidad en uno de los dos sistemas distribuidos desarrollados por el departamento de Desarrollo Transversal de Telefónica I+D: PopBox y Rush. El diseño y el desarrollo de esta solución deberá tener en cuenta las características de estos dos sistemas, como el uso de una base de datos NoSQL (Redis), y un servidor Nodejs.

Concretamente, los objetivos que se han alcanzado a la finalización de este proyecto son los siguientes:

1. Control de versiones del código implementado: el desarrollo del proyecto se incluirá en el flujo de trabajo principal de PopBox, por lo que he necesitado adquirir conocimientos sobre software de control de versiones (Git), y manejo de una forja de desarrollo (GitHub).

2. Diseño e implementación de una solución de escalabilidad: se realizó un diseño usando algoritmos de hash consistente. Específicamente, se utilizó la estrategia de nodos virtuales implementada por Amazon en sus sistema Dynamo. El sistema podrá distribuir la carga entre distintas bases de datos Redis para aumentar su capacidad de almacenamiento global sin sufrir por ello una disminución en el rendimiento o el tiempo de respuesta.

[2]Propiedad deseable de un software que indica su habilidad para extender el margen de operaciones sin perder calidad.

[3]Propiedad de un sistema de poder atender a los cambios de demanda de una manera ágil y dinámica.

3. Diseño e implementación de una solución de elasticidad en el escalado del sistema: se ha implementado un servidor de configuración distribuido que permite aumentar o disminuir la capacidad de almacenamiento de claves de PopBox. En esta fase se ha tenido muy en cuenta la fiabilidad de la solución por ser un punto crítico en cuanto a consistencia de datos.

4. Diseño de una solución de alta disponibilidad: se ha diseñado e implementado una solución haciendo uso de la funcionalidad de monitores Redis y replicación maestro/esclavo. Esta solución permite que el servicio no quede interrumpido en caso de producirse algún fallo en alguno de los componentes, además de permitir la recuperación de datos en caso de perdida de los mismos.

5. Evaluación de las funcionalidades de escalabilidad, elasticidad y alta disponibilidad descritas anteriormente: se realizaron tests unitarios y tests de integración para evaluar el comportamiento del sistema en diferentes situaciones. Además se han realizado "monkey tests"[Inc11], tests que simularán escenarios aleatorios tomando como variables el número de servidores activos, número de bases de datos, tiempos de fallo del servidor...

6. Evaluación del rendimiento de la solución planteada: análisis del rendimiento mediante tests end-to-end utilizando herramientas ad-hoc y software de medición del rendimiento como Apache Bench (ab).

1.3. Estado anterior de los sistemas PopBox y Rush

El estado de estos dos sistemas antes del inicio de este proyecto era de software como producto. PopBox y Rush eran componentes listos para ser instalados en un conjunto de máquinas, pero no proporcionaban garantías suficientes como para poder ser consumidos como servicio. La evolución natural de estos dos productos obligaba a proporcionar un servicio escalable y altamente disponible para proporcionar seguridad y fiabilidad a los potenciales clientes.

En el estado de componente de PopBox y de Rush, el sistema distribuía la carga de almacenamiento entre distintos nodos Redis, pero no era posible atender a una mayor demanda de capacidad de almacenamiento que la ofrecida por el despliegue inicial. Por lo tanto, el sistema **no era escalable**. A la finalización de este proyecto se ha conseguido que el sistema pueda atender a una demanda creciente sin perder rendimiento, así como minimizar recursos en momentos de baja carga en el sistema.

Así mismo, estos dos sistemas no proporcionaban una manera de aumentar la capacidad del sistema, en términos de almacenamiento y potencia, de una manera flexible. Una nueva configuración requería del rearranque del sistema al completo, con la consecuente perdida temporal del servicio. El sistema en su fase de componente **no era elástico**. El diseño mostrado en el presente trabajo permite la inserción o

eliminación de recursos mientras el sistema esta en funcionamiento, evitando interrupciones de servicio que deterioran el valor ofrecido por SaaS.

Por último, ambos sistemas planteaban una solución poco eficiente para proporcionar **alta disponibilidad**, por lo que no estaban preparados para su uso como SaaS. Un fallo en alguno de los nodos en un clúster suponía la redirección de todas las peticiones a un nuevo clúster. Esta solución requería replicar el sistema completo varias veces, con el consiguiente desperdicio de recursos que esto producía. En la solución planteada en este proyecto, se puede asegurar un funcionamiento altamente disponible sin necesidad de copiar el sistema completo, simplemente replicando puntos susceptibles a fallo y manteniendo una redundancia de datos dentro de cada clúster.

Capítulo 2

Tecnologías utilizadas

En este capítulo se mostrarán las tecnologías sobre las que se ha desarrollado el presente trabajo, así como las herramientas más destacables que se han usado a lo largo de este.

2.1. NodeJS

Tanto PopBox como Rush utilizan esta plataforma para ofrecer su servicio. El alto rendimiento y la escalabilidad de estos dos sistemas son debidos en parte al rendimiento que proporciona NodeJS.

NodeJS es una plataforma de programación en el lado del servidor basada en Javascript. Todo el entorno Javascript de NodeJS corre sobre el motor desarrollado para Google Chrome, Javascript V8[Abe11]. El modelo que utiliza NodeJs es orientado a eventos, esto es, el flujo del programa no viene impuesto por el programado sino por los eventos que suceden en el servidor; y asíncrono, las acciones que se realizan no siguen un intervalo fijo. NodeJS fue concebido para diseñar aplicaciones de red altamente escalables, ya que este implementa una pila de protocolos de red como TCP, UDP y HTTP.

El gran rendimiento de esta plataforma en el sentido de servidor de red permite manejar decenas de miles de conexiones concurrentes. Un servidor Java o un servidor Apache necesita crear un nuevo hilo de ejecución por cada conexión abierta con algún cliente. Esta infraestructura requiere de mucha memoria (aproximadamente 2MB) por cada conexión establecida, limitando un servidor de 8GB a 4000 conexiones simultaneas[Abe11]. Javascript es mono-hilo y orientado a eventos, por lo que NodeJs no crea un nuevo hilo con cada conexión establecida, sino que este dispara una ejecución de evento dentro del proceso servidor. Además, NodeJS no queda bloqueado en operaciones E/S[1], ya que en el caso de este tipo de operaciones NodeJS creará otro hilo para realizar dicha operación, y por lo tanto nunca se bloqueará el proceso.

[1]Entrada/Salida

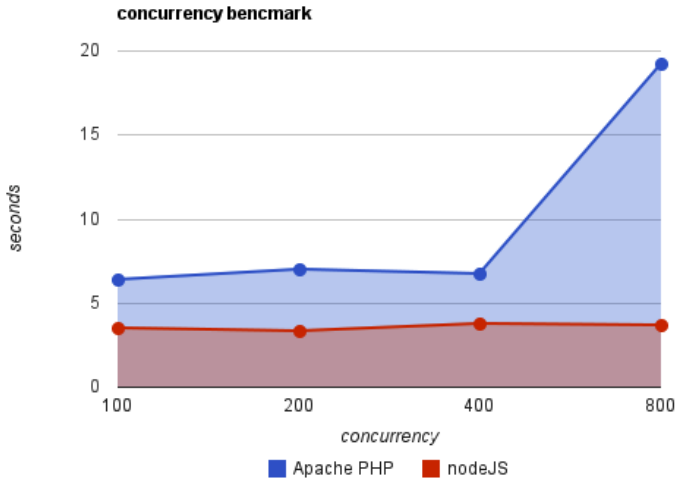

Figura 2.1.1: Comparativa del tiempo de respuesta de Apache y Node dependiendo del número de conexiones concurrentes.

Al contrario que Apache, NodeJS no es un framework que se instale en un servidor y se pueda poner en funcionamiento, sirviendo contenido contenido estático y dinámico como PHP. Las aplicaciones en node han de ser diseñadas usando módulos de la propia plataforma o módulos externos.

2.1.1. Módulos

Al igual que en otros servidores como Apache o Django, NodeJS permite ampliar su funcionalidad mediante módulos. Algunos de estos módulos vienen incluidos en el código fuente de la aplicación, como los módulos HTTP, STDIO, OS... que permiten realizar las funciones básicas de un servidor.

Además de estos módulos incluidos en el sistema, NojeJS permite utilizar módulos externos de una manera muy sencilla. Estos módulos extienden las funcionalidades de las aplicaciones de una manera muy amplia, facilitando tareas como la creación de WebSockets o la conexión con bases de datos. Esta inclusión de módulos en las aplicaciones sigue la especificación de CommonJs: 'require' importa los módulos, y 'export' exporta funciones de la API de un módulo concreto.

NodeJS viene provisto de un gestor de paquetes, "node package manager" (npm), que permite instalar módulos externos en el sistema y gestionar las dependencias de los módulos del sistema. Es similar al gestor de paquetes de Ruby, RubyGems.

Algunos módulos externos de NodeJS son parte fundamental de los componentes PopBox y Rush. Los más importantes son los siguientes:

Express Es un framework para construir aplicaciones web similar a Django en Python o Rails en Ruby. Ofrece manejo de rutas, middleware, manejo de sesión... En los proyectos de PopBox y Rush, Express proporciona una interfaz REST.

Async Este módulo ofrece un conjunto de funciones para trabajar fácilmente de manera asíncrona. Algunas de las funciones que ofrece son map, reduce o filter, pero también ofrece funciones para manejar flujos asíncronos de una manera sencilla. Este módulo esta presente a lo largo de los sistemas PopBox y Rush, manejando algunos flujos de ejecución asíncronos.

Node_redis Es un cliente de Redis en NodeJs. Soporta todos los comandos de Redis, así como subscripciones a canales Pub/Sub.

2.2. Redis

El sistema de persistencia de los componentes PopBox y Rush es Redis, lo que proporciona un acceso muy rápido a los datos y permite muchas conexiones concurrentes desde los diferentes Agentes del sistema.

Redis es una base de datos NoSQL. NoSQL (Not only SQL), es un nuevo tipo de almacenes de datos no relacionales. Las bases de datos no relacionales no siguen esquemas y suelen utilizar un almacenamiento del tipo clave-valor y orientado a documentos. Estas bases de datos no garantizan completamente todas las características ACID (atomicidad, coherencia, aislamiento y durabilidad). En compensación, estos sistemas son muy escalables horizontalmente, manejan enormes cantidades de datos y no generan cuellos de botella en los sistemas. Es por eso que con el crecimiento de la Web y la aparición de la Web2.0, este tipo de bases ha proliferado. Las bases de datos tradicionales (SQL) no podían procesar la ingente cantidad de datos que se generaba en la nueva Web. La estructura horizontal de estos conjuntos de datos permitía ser procesada en una base de datos sin esquema. Las principales compañías de Internet como Amazon, Google, Facebook o Twitter encontraron en las bases de datos no relacionales una solución para el tratamiento de la enorme cantidad de datos que estas generaban.

Lo más destacable de Redis es que utiliza un almacenamiento en memoria. Esto permite que las operaciones de lectura y escritura se realicen de una forma asombrosamente rápida. Redis es capaz de soportar cientos de miles de transacciones por segundo, aportando un rendimiento altísimo. A pesar de ser una base de datos en memoria, Redis permite el almacenamiento persistente.

2.2.1. Estructura de datos

La diferencia con el sistema memcached (que también utiliza almacenamiento en memoria) es que Redis es un sistema de almacenamiento clave-valor. Esto quiere decir que los datos almacenados poseen estructura. Los diferentes tipos de estructuras de Redis son los siguientes:

- Cadenas de caracteres (Strings)

- Conjuntos (Sets)
- Listas (Lists)
- Conjuntos ordenados (Sorted Sets)
- Tablas Hash (Hashes)

Este conjunto de estructuras dota de muchísima flexibilidad al sistema, permitiendo realizar estructuras globales muy complejas.

2.2.2. Pub/Sub

La base de datos permite comunicación entre clientes mediante sistemas de publicación y subscripción y colas de mensajes. Esto permite a los clientes interactuar mediante este medio y crear aplicaciones en tiempo real como chats, juegos... En este proyecto se ha hecho uso de esta funcionalidad para realizar el sistema de comunicación para la sincronización. Los diferentes Agentes se comunican con los servidores de configuración mediante este sistema.

2.2.3. Replicación Maestro/Esclavo

Redis permite replicación maestro/esclavo. Esto permite mantener redundancia de datos en varias instancias Redis para proveer al sistema de alta disponibilidad. La escritura en una instancia maestra se propaga a todos los esclavos de esta. Además, un nodo esclavo puede a su vez ser maestro de otro nodo, pudiéndose crear cadenas maestro esclavo para delegar la carga de las sincronizaciones en los esclavos y mejorar el rendimiento del sistema.

La replicación maestro esclavo ha sido usada en el sistema PopBox para proporcionar un mecanismo de sincronización dentro de un mismo clúster. Estos mecanismos, junto con el sistema de monitores de Redis se han utilizado para dotar a PopBox de alta disponibilidad mediante redundancia y promoción de esclavos.

Otra ventaja de este sistema es la de poder delegar las lecturas en los diferentes nodos esclavos, de manera que solo las escrituras se realicen en el nodo maestro y liberando a este de carga. Al poder poseer un nodo maestro varios nodos esclavos, la carga de lecturas se podrá dividir entre los diferentes esclavos, proporcionando una velocidad de lectura muy superior. Este caso no siempre puede ser una opción. En sistemas donde la escritura de un dato y su posterior lectura están muy cercanas temporalmente esta estrategia puede dar lugar a fallos. Esto es porque la propagación de un dato desde un maestro a sus diferentes esclavos no es instantánea. Es decir, en cierto momento, un dato contenido en el nodo maestro puede no existir aún en alguno de sus nodos esclavos.

2.2.4. Otras características

Además de las funcionalidades mencionadas en los puntos anteriores, Redis provee otras funcionalidades muy útiles en determinadas aplicaciones. Algunas de estas son:

- Espiración de claves: Redis permite fijar un periodo de existencia de una clave. Pasado este periodo, la clave expirará automáticamente y de eliminará de la base de datos. Esto es necesario en sistemas de caché.
- Transacciones: Redis posee un motor de ejecución de scripts en Lua dentro de su núcleo. Las ejecuciones de estos scripts son atómicas, por lo que se pueden realizar operaciones sobre varias claves dentro del propio Redis, sin tener que realizar varias llamadas a este.
- Persistencia: Redis permite almacenar los datos de forma persistente. Esta opción es altamente configurable, pudiendo fijar el tiempo entre salvados o la cantidad de datos que se deben escribir antes de realizar dicho salvado.
- Monitores: En la última versión de Redis aparece el concepto de monitores. Este concepto se materializa mediante los Sentinels. Un Sentinel monitorizará un nodo maestro y a sus esclavos y podrá disparar un mecanismo de failover en caso de fallo de la base de datos padre. Estos Sentinels son una solución, junto con la replicación maestro/esclavo para proveer HA.

2.2.5. Futuro de Redis

Según el blog del creador de Redis, Salvatore Sanfilippo, el próximo reto que está afrontando Redis es Redis Cluster[San13]. Redis Cluster proporcionará al sistema escalabilidad mediante sharding y alta disponibilidad mediante el sistema de Sentinels. Toda la distribución de claves, redistribución y monitorización de nodos se realizará a nivel interno, evitando la responsabilidad que actualmente tiene el usuario de realizar todas estas funciones.

Capítulo 3

Descripción de los sistemas PopBox y Rush.

3.1. PopBox

PopBox es un sistema de buzones distribuido de alto rendimiento. Su principal función es la de proporcionar buzones de mensajes (transacciones) a un gran número de usuarios y en un tiempo mínimo.

PopBox está preparado para almacenar buzones seguros protegidos por contraseña gracias al uso de https y basic-auth. La API de PopBox es íntegramente REST[1], lo que permite mantener todo el estado de la aplicación en el servidor y ser consumido por cualquier cliente que implemente el protocolo HTTP[Fie00].

Los elementos principales de PopBox son las colas y las transacciones.

- Colas: una cola o buzón es un contenedor de transacciones. Las colas pueden ser públicas o privadas. Las operaciones más habituales son las de PUSH (insertar mensajes en una o varias colas) o POP (extraer cierto número de mensajes en orden de inserción). También existen operaciones de observación, pero no son el principal fin de PopBox.

- Transacciones: es la información que se quiere almacenar y recuperar. Una transacción puede estar contenida en uno o más buzones y tener prioridad alta o baja. Los campos de una transacción son:

  Payload: el contenido principal de la transacción que contiene la información importante. Es un String de longitud máxima prefijada y su presencia en la transacción es obligatoria.

  Callback: en el momento de hacer un POP sobre cierta transacción, PopBox realizará una petición POST a la URL indicada en el campo callback. El

[1]REST : (Representational State Transfer)

cuerpo de la petición será la transacción en sí. Es un campo opcional de tipo String.

Queue: es el campo que define las colas o cola que contendrán la transacción. Este campo es un Array de objetos y su presencia es obligatoria.

Priority: la prioridad de la transacción. Los únicos valores permitidos son«H» o«L», alta y baja respectivamente. Este campo se tendrá en cuenta a la hora de extraer las transacciones de las colas, teniendo los mensajes con prioridad«H» mayor prioridad de salida que aquellas con prioridad«L». Su presencia es obligatoria.

expirationDelay: el tiempo (en segundos) tras el cuál la transacción expira. Es un campo opcional, y si no se define el tiempo de expiración será el predeterminado en la configuración.

3.1.1. Arquitectura de despliegue

La arquitectura de PopBox está compuesta por tres elementos: los agentes, las bases de datos transaccionales, y la base de datos de logging.

Un agente es el punto de entrada a la aplicación. Se trata de una aplicación REST que almacena y extrae las transacciones de la Base de Datos siguiendo unas ciertas lógicas. En un despliegue real existirá más de un agente, ya que el sistema es escalable.

Las bases de datos transaccionales son las que es encargan de almacenar todas las transacciones PopBox. Como se explicó anteriormente, existen dos tipos de elementos lógicos: las colas y las transacciones. En la arquitectura de PopBox, las colas se almacenan en diferentes bases de datos según un algoritmo de hashing; y las transacciones se almacenan en una única base de datos.

La base de datos de logging se encarga de almacenar el histórico de inserciones y extracciones durante una sesión de PopBox, es una única instancia BD.

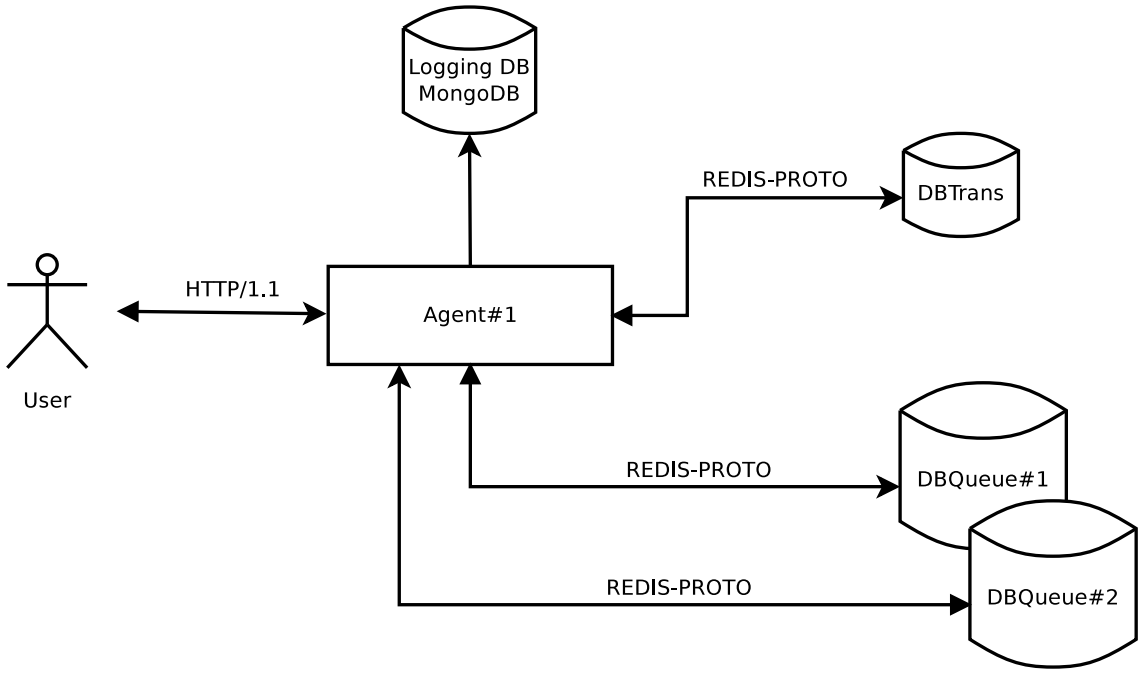

Figura 3.1.1: Arquitectura de despliegue de PopBox

3.1.2. Arquitectura lógica

La arquitectura lógica del agente está dividida en varios módulos que se encargan de realizar diferentes tareas. Además, PopBox sigue el paradigma AOP para el logging. Los módulos más importantes son los siguientes:

- Interfaz REST: Es el punto de entrada a la aplicación. Se definen todas las operaciones HTTP (GET, POST, DEL...), para acceder a todas las funcionalidades de PopBox. Por defecto, PopBox levanta el servicio en dos puertos distintos, uno HTTP y otro HTTPS.

- Lógica: Realiza las operaciones sobre el cuerpo de la petición y devuelve el resultado resultante de dicha operación.

- Lógica DB: Se encarga de realizar operaciones sobre las bases de datos para insertar y recuperar el contenido de las transacciones y de las colas.

- Cluster DB: Dirige las operaciones de la lógica a determinada base de datos, balanceando la carga entre todas las disponibles.

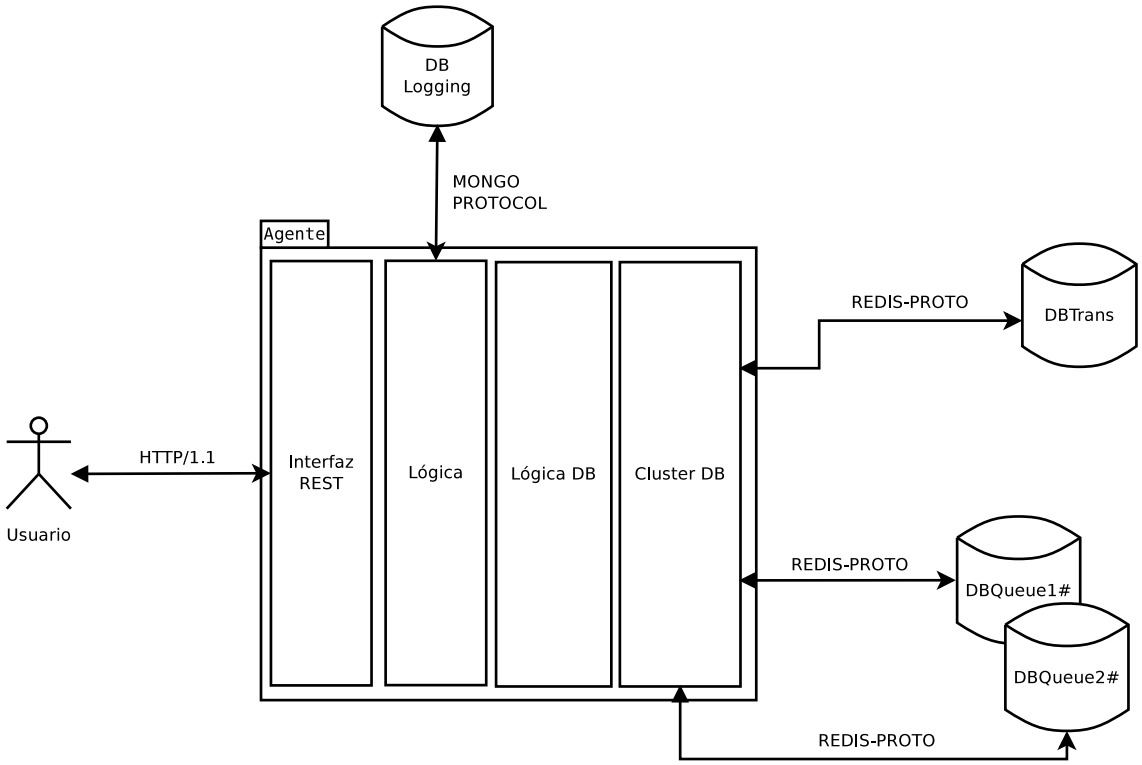

Figura 3.1.2: Arquitectura Lógica de PopBox

3.2. Rush

3.2.1. Descripción general

Rush es un proxy asíncrono distribuido. Con Rush se pueden realizar peticiones asíncronas a un servidor sin esperar la respuesta, y consultar dicha respuesta más adelante. El sistema es completamente escalable ya que se pueden añadir y eliminar nodos del sistema de manera elástica dependiendo de la carga de trabajo.

El sistema permite realizar diferentes tipos de peticiones mediante el uso de diferentes políticas.

3.2.2. Políticas de petición

- Política OneWay: la petición que realiza el usuario no recibe la respuesta del servidor destino.

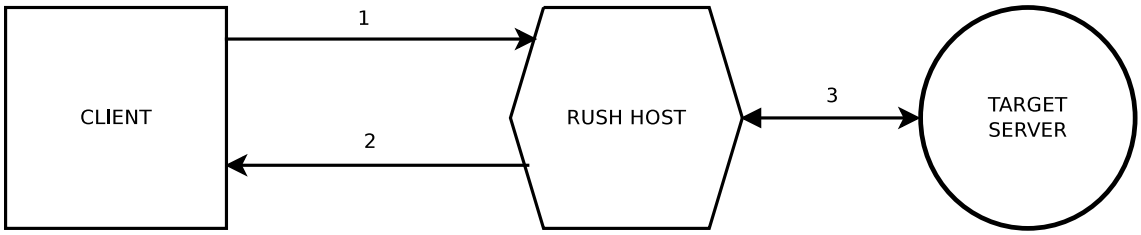

Figura 3.2.1: Política OneWay

- Política Persistence: la respuesta asignada a la petición es almacenada. El usuario puede consultar dicha respuesta en cualquier momento mediante el identificador de petición.

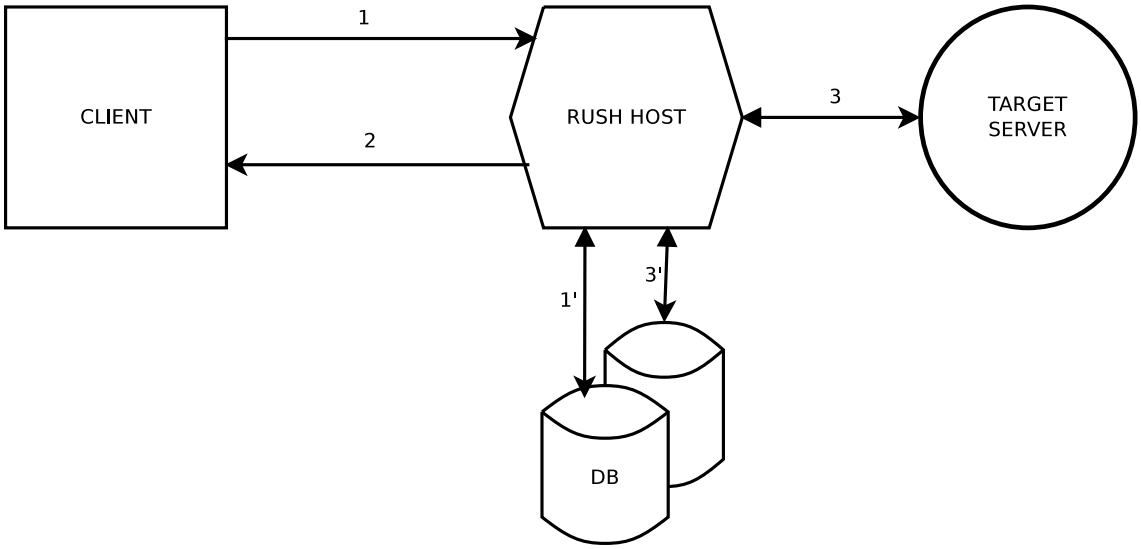

Figura 3.2.2: Política Persistence

- Política Retry: permite el reintento automático de una petición en el caso de que está no haya recibido una respuesta válida. Es usuario puede definir el número de reintentos y su intervalo.

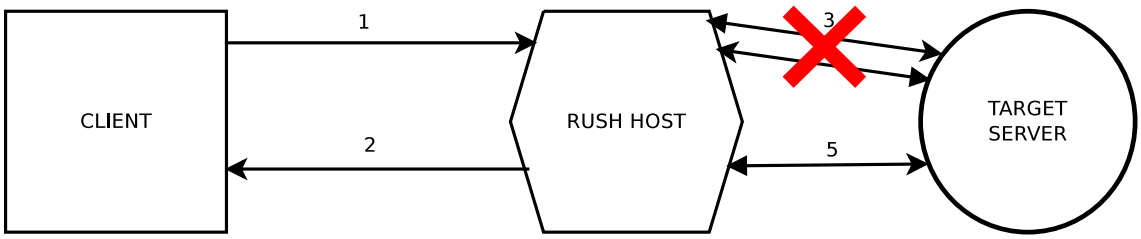

Figura 3.2.3: Política Retry

- Política Callback: permite redirigir la respuesta del servidor asociado a otro servidor. De este modo, el cuerpo de la respuesta de la petición original, se convierte en el cuerpo de la segunda petición.

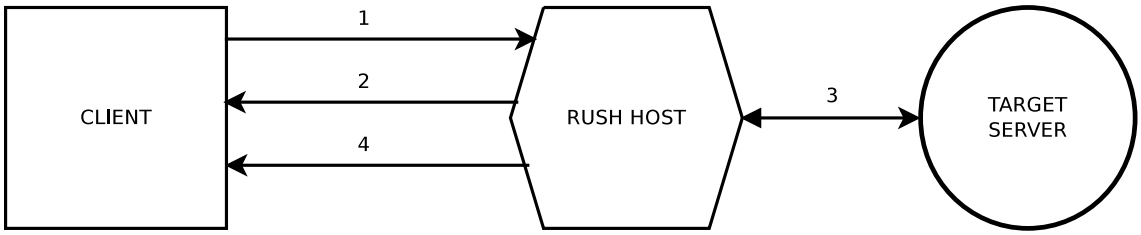

Figura 3.2.4: Política Callback

Rush permite la combinación de todas estas políticas permitiendo aprovechar las ventajas de cada una. Por ejemplo, se podría combinar la política Callback con la política *Persistence* si se quiere reintentar una petición hasta que sea exitosa, y en caso de éxito almacenar la respuesta para su posterior consulta.

3.2.3. Arquitectura interna

El diseño de Rush se basa en Node.js cómo plataforma de servidor, y en Redis como sistema de almacenamiento y comunicación. Rush es un proxy, por lo que el almacenamiento es principalmente de tipo transaccional, lo que hace de Redis una base de datos idónea para este sistema.

La arquitectura de Rush se basa en dos actores, el *listener* y el *consumer*. Un despliegue Rush puede estar formado de varios *listeners* y *consumers.*

- *Listener*: la comunicación con el usuario se realiza a través de este componente. Está dotado de una interfaz HTTP, a través de la cual se realizan las peticiones y se obtienen las respuestas almacenadas (en el caso de Política Persistence). Este componente almacena las peticiones recibidas por parte de cualquier cliente y delega el trabajo de realizar la petición al servidor destino a los *consumers.*

- *Consumer:*es el encargado de realizar la petición al servidor destino. Los consumers esperan a que haya alguna petición encolada, y en el momento de encontrar alguna simplemente la realizan siguiendo alguna de las políticas especificadas.

La ventaja de Rush es que no es necesario que exista ningún conocimiento entre *listeners* y *consumers*, ya que la comunicación entre ellos se realiza a través de Redis mediante un sistema de colas.

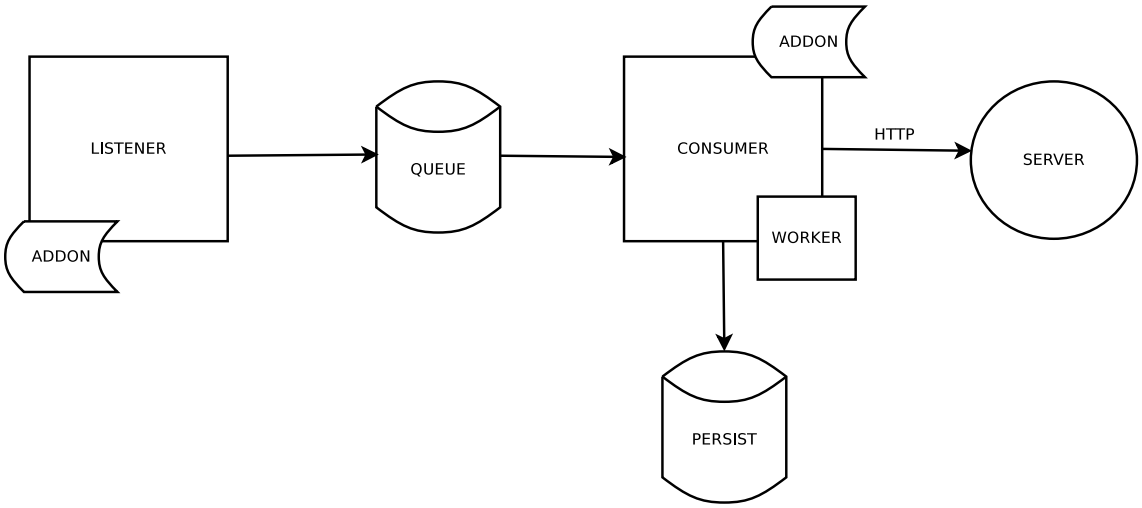

Figura 3.2.5: Arquitectura lógica de Rush con un sólo *listener* y un *consumer*

Conociendo el funcionamiento de Rush, se puede plantear el siguiente escenario:

1. El cliente realiza una petición http a Rush, que será respondida con un identificador de petición, que podrá ser utilizado posteriormente. El *listener* se encarga de encolarla en su respectiva cola Redis, en la cual se encuentran las peticiones aún no asignadas a ningún *consumer.*

2. El *consumer* se mantiene conectado a la cola de peticiones pendientes. Al encontrarse con una petición pendiente, este la extrae y la asigna a un *worker.* El *worker* es el encargado de realizar la petición siguiendo la política pertinente. Una vez este ha realizado correctamente la petición y ha recibido una respuesta, se lo notifica al *consumer*, que se encarga de destruirlo. Un *consumer* puede generar varios *workers* activos. El número de *workers* concurrentes es un parámetro de configuración, por lo tanto puede haber peticiones pendientes que no se atiendan al instante.

3. El *consumer* se encargará de almacenar la respuesta de la petición para su posterior consulta por parte del cliente. La consulta se realiza de manera asíncrona, haciendo una petición http con la id de petición a Rush.

La arquitectura de Rush permite la incrustación de add-ons para añadir algunas funcionalidades al sistema. Este sistema esta basado en la publicación de ciertos eventos de error o de estado relacionados con el procesamiento de peticiones. Por ejemplo, permiten saber si una petición está pendiente, se ha completado, ha fallado o está en proceso. Para almacenar estos eventos se utiliza la base de datos MongoDB.

Capítulo 4

Estado del arte

4.1. Escalabilidad

La escalabilidad es la habilidad que tiene un sistema de adaptarse a un crecimiento de carga sin perder calidad en los servicios ofrecidos. Normalmente la escalabilidad se realiza hacia arriba, es decir, aumentando el tamaño y la potencia del sistema; pero también puede se hacia abajo, utilizando solo los recursos necesarios en situaciones de baja demanda.

Existen dos maneras de escalar un sistema: "scale in" y "scale out".

El "scale in" o escalado vertical consiste en el incremento de recursos en un solo nodo, ya sea añadiendo memoria, capacidad de procesamiento .etc. El principal problema que existe con el escalado vertical es que es el coste de aumentar los recursos en un solo nodo es exponencial y siempre tiene un límite.

El "scale out" o escalado horizontal consiste en el incremento de los nodos en un sistema, de tal manera que al aumentar la demanda de capacidad o de proceso, se añaden más nodos al sistema, sin tener que ampliar la capacidad de estos. El concepto de escalabilidad horizontal esta basado en la idea de que, cuando la cantidad de datos en un nodo y las peticiones hechas a este crecen linealmente, el tiempo de respuesta de cada petición crece exponencialmente. También crece exponencialmente el coste de crear y mantener nodos más grandes y potentes.

En el caso de las bases de datos, la escalabilidad horizontal se ha de llevar a cabo separando la cantidad total de datos en diferentes instancias, de manera que cada instancia posea un rango datos determinado.

4.1.1. Replicación Maestro/Esclavo

Es el sistema utilizado por muchas organizaciones. El funcionamiento del sistema es simple: las escrituras se realizan en el maestro y este replica automáticamente todos los datos a cada uno de los esclavos presentes en el sistema. Las operaciones

de escritura (CREATE, UPDATE, DELETE) se realizan en el servidor, mientras que las de lectura se realizan en los esclavos. La escalabilidad se obtiene con un balanceador de carga que reparta las operaciones de lectura entre los distintos esclavos. Esta estrategia es adecuada para sistemas en los que se realizan pocas escrituras y muchas operaciones de lectura, pero en sistemas donde esto no ocurre encontramos los siguientes problemas:

- La existencia de un solo nodo para escrituras limita la escalabilidad. A cierto nivel de carga se producirán cuellos de botella.
- La replicación hacia cada esclavo no es inmediata, por lo cual no es asegurable el tener una copia actualizada de la base de datos en cada uno de los esclavos.
- En el caso de utilizarse este sistema para ofrecer escalabilidad así como alta disponibilidad, el problema anterior puede resultar fatal. Si el nodo maestro falla, la latencia de escrituras en los esclavos resulta en la perdida de los datos desactualizados en estos.

4.1.2. Sharding

La palabra 'shard' significa literalmente trozo. El sharding es la división de una base de datos en otras más pequeñas, por lo tanto más manejables. El término 'sharding' fue utilizado por primera vez por los ingenieros de Google, popularizado gracias a la publicación de un artículo sobre BigTable[FC06]. El sharding sigue el concepto de «shared-nothing», cada nodo es completamente independiente de los demás, no tiene que interactuar con los demás nodos y los datos contenidos en este son completos. Esta aproximación posee numerosas ventajas, tanto en escalabilidad como en alta disponibilidad:

- Alta disponibilidad: en el caso de que un nodo falle, los demás nodos podrán seguir ofreciendo servicio.
- Peticiones más rápidas: menos datos significa menos tiempo de inserción y extracción.
- Mayor ancho de banda: poseer diferentes instancias ofrece la capacidad de realizar escrituras paralelas en diferentes servidores. El principal problema de muchos sistemas es la formación de cuellos de botella en las operaciones de escritura.
- Paralelismo: la capacidad de trabajar simultáneamente en varias bases de datos permite realizar diferentes tareas en paralelo, lo que conlleva mayor rendimiento.

Para que los puntos anteriores se cumplan, hay que tener en cuenta una serie de consideraciones[Cod]:

- Fiabilidad: este es un punto que se ha de tener en cuenta en cualquier sistema. Las bases de datos son un punto crítico de cualquier sistema y no deben ser propensas a fallos. Hay que llevar a cabo una serie de acciones para garantizar la total disponibilidad de todos los shards del sistema:

  - Backups automáticos de cada BD.
  - Redundancia de cada instancia. Para asegurar la integridad de los datos, cada instancia tiene que tener dos replicas en funcionamiento.
  - Detección de fallos y recuperación automática.
  - Gestión de errores.

- Distribución de las peticiones: en el apartado anterior hemos hablado del mayor ancho de banda que se puede conseguir gracias al sharding. Para aprovechar está ventaja es necesario que las peticiones a cada shard se realicen de manera uniforme al conjunto. Si no existe una uniformidad en la distribución, es posible que se produzcan cuellos de botella en los nodos más demandados. Además, con una distribución uniforme, se aprovecha mejor la capacidad de almacenamiento de cada nodo.

- No interdependencia entre shards: para garantizar alta disponibilidad es necesario que cada nodo sea una instancia independiente de las demás. En una base de datos distribuida, los inner-join's deben ser evitados, y en su lugar utilizar tablas globales que contengan los datos que vayan a ser requeridos por todas las instancias.

Estrategias de sharding

Como se ha visto en el apartado anterior, la distribución uniforme de claves es un aspecto fundamental del sharding, ya que evita cuellos de botella en las escrituras y permite aprovechar mejor el espacio de cada nodo

En algunas ocasiones, lo que se busca es dirigir los datos a una determinada BD atendiendo a un valor. Por ejemplo, Twitter almacena todas las tablas de un usuario en una sola base de datos. También puede interesar distribuir los datos atendiendo a la situación geográfica del usuario para evitar latencia en la comunicación. La distribución por campo, como cabe esperar, no es uniforme.

La clave que se utiliza para decidir la distribución entre las bases de datos se denomina 'shard key' y normalmente es el hash de la combinación de varios campos de la fila.

Existen diferentes estrategias para distribuir estas claves entre las diferentes BD del sistema:

- Por clave: existe una estructura clave-valor en la que la clave es el shard-key y el valor es el nodo al que se han de dirigir los datos. El problema de esta distribución es que no es elástica; cada vez que se añade un nodo hay que rehacer completamente la estructura clave-valor.
- Por módulo: la elección del nodo se realiza calculando el módulo de la clave con el número de nodos actuales. Es la aproximación más pobre ya que, como se verá más adelante, es muy poco elástico.
- Hashing consistente: cada nodo posee un espacio de claves, por lo tanto, los datos irán dirigidos al nodo que posee la clave actual. Es el algoritmo más eficiente. En el caso de añadir un nuevo nodo al sistema, sólo se distribuiránK/n claves, donde$K$ es el número total de claves en el sistema, yn el número total de nodos[Dav97].

Distribución por módulo En la distribución por módulo, también llamada"modula", el cliente selecciona la BD primero calculando el hash, y después calculando el modulo de ese hash con el número de servidores.

$$DB = h(k) \,\%\, n$$

$h(k)$ es el hash de la clave o claves, y n es el número de servidores actuales. Esta configuración es apropiada para configuraciones estáticas, donde no se espera añadir ninguna BD durante el despliegue del sistema. En el caso de añadirse un nuevo servidor, la distribución de las claves en cada uno de los servidores sería completamente distinta, teniendo que redistribuir casi todos los datos o, en el caso de BD de caché volver a generar todas las caches. Proponemos el escenario siguiente:

1. Tenemos un array claves = [1,2,3,4,5] y un número de servidores n = 2.
2. Calculamos el servidor destino de cada una de ellas de acuerdo al siguiente algoritmo:

Algorithm 4.1 Cálculo de base de datos de destino usando 'modula'

Require: n = número de bases de datos, h(k) = hash de k;

```
for all clave in claves do
    db = h(clave) % n
    imprimir db
end for
```

3. El resultado del calcular cada uno es: BDs destino = [1,0,1,0,1], siendo 0 y 1 las dos diferentes bases de datos disponibles.
4. Volvemos a calcular el servidor destino de cada clave, pero en este caso el número de BD's será n = 3;

5. El resultado del calcular cada uno es: BDs destino = [1,2,0,1,2], siendo 0, 1 y 2 las tres diferentes bases de datos disponibles.

Podemos observar que la distribución de claves en cada caso es completamente distinta, produciéndose una distribución distinta en el 80 % de los casos.

Se concluye que esta es una estrategia de distribución poco apropiada para configuraciones dinámicas, en las que se pueden agregar nodos al despliegue en un momento dado.

Distribución por hash consistente La idea de este algoritmo de distribución fue publicada en 1997 por Karger et al.[Dav97]. Fue desarrollada en el MIT en un intento de mejorar los algoritmos de caching distribuido. Actualmente es utilizado por muchos sistemas de almacenamiento distribuido tales como$D$ynamo o$m$emcached. Esta aproximación es muy eficiente ya que al añadir o eliminar un nodo la redistribución de datos será deK/n claves, donde$K$ es el número total de claves en el sistema, yn el número total de nodos[Dav97].

Primera aproximación Para entender como funciona el hash consistente imaginemos un anillo con valores distribuidos en el intervalo $[0, 1)$.

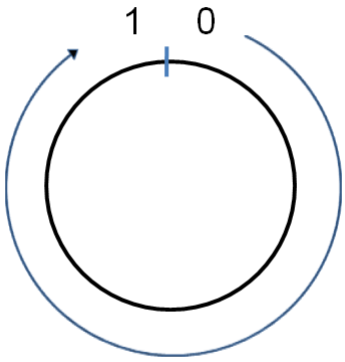

Figura 4.1.1: Anillo de hashing$[0, 1)$

Suponiendo que tenemos un número de máquinas $0, \ldots, n-1$. Si la función de hashing, por ejemplo MD5, tiene un intervalo de valores $[0, R)$, entonces el circulo tendrá un intervalo de valores del mismo rango. Cada máquina en el rango$j = 0, ..., n-1$ se asignará, a través de su función de hashing, a un punto$hash(j)$ del anillo de hashing[Nie09]. Para un $n = 3$, este sería el aspecto:

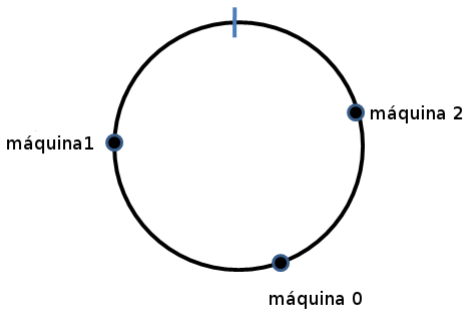

Figura 4.1.2: Máquinas distribuidas en el intervalo$[0, R)$

El espacio de claves correspondiente a cada máquina es aquel que se encuentra entre la clave asignada a esta máquina y la clave de la máquina menor más próxima, es decir, el espacio de claves asignado a "máquina1" sería el espacio entre "máquina1" y "maquina0". Teniendo esto en cuenta, se calcula el hash de la clave que queremos introducir.

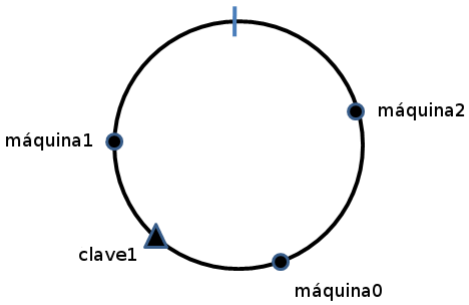

Figura 4.1.3: clave hasheada en el espacio$[0, R)$

La clave "clave1" pertenece ahora a "máquina1", como se puede observar en4.1.3, porque esta contenida en su espacio de claves. La efectividad de la estrategia de distribución por hash consistente se observa a continuación. Se añade una máquina más al sistema, a la que llamaremos "máquina3".

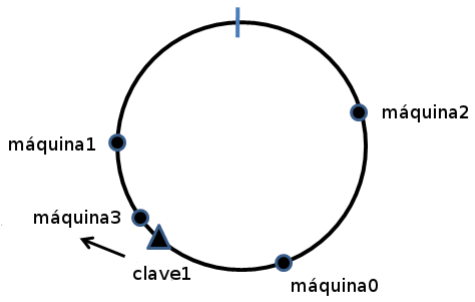

Figura 4.1.4: nueva máquina en el espacio de claves

El espacio de claves de “máquina3” será ahora el comprendido entre “máquina3” y “máquina0”. La clave “clave1” pertenece ahora a este espacio de claves, por lo tanto se redistribuirá hacia ésta. Se observa que el único espacio de claves afectado es el que anteriormente pertenecía a “maquina1” y por lo tanto la redistribución de claves es mínima, aproximadamenteK/n claves, donde$K$ es el número total de claves en el sistema, y n el número total de nodos[Dav97].

Segunda aproximación En el anterior apartado hemos dado por hecho algunos aspectos:

- Que el algoritmo de distribución de claves ofrece una distribución completamente uniforme: En todos los algoritmos de hashing la distribución se aproxima a ser completamente uniforme según aumenta el número de claves, pero, en el caso de poseer pocas máquinas, se puede producir una mala distribución, con la consecuente mala distribución de carga entre los nodos.
- Que todas las máquinas asignadas a los nodos del anillo de hashing poseen la misma capacidad y, por consiguiente, han de recibir la misma carga. Un entorno real es heterogéneo, es decir, dispondremos de diferentes máquinas con diferentes capacidades, por lo tanto es necesario distribuir la carga del sistema atendiendo a las capacidades de cada nodo.

Ninguna de las dos afirmaciones anteriores, como observamos, es correcta.

Para solventar los dos problemas anteriores es necesario introducir el concepto de nodos virtuales o tokens. Un nodo virtual es un alias de un nodo real (una BD). Un nodo virtual pertenecerá a un único nodo real, pero un nodo real podrá poseer muchos nodos virtuales; la relación entre nodos reales y nodos virtuales es (1,n). La finalidad de los nodos virtuales es distribuir la carga uniformemente a lo largo de todo el espacio de claves.Mientras con una aproximación sin nodos virtuales, un nodo pertenece a un único valor en el anillo de hashing y puede poseer un espacio de

claves más o menos grande; los nodos virtuales se distribuyen a lo largo de todo el anillo, y la suma de sus respectivos espacios de claves es aproximadamente $\frac{a_x}{\sum_{i=1}^{N} a_i}$ del espacio de claves completo, donde N es el número total de nodos, a_x es el número de representantes del nodo actual, y a_i el número de representantes de cada nodo en el sistema.

Como vemos, este sistema tiene dos ventajas:

- Distribución uniforme de carga a lo largo de todo el anillo.
- Distribución de carga de cada nodo atendiendo a su peso, es decir, a la capacidad del mismo.

Existen varias estrategias a la hora de distribuir los nodos virtuales a lo largo del anillo de hashing atendiendo a diferentes aspectos de rendimiento[Ove12].

Asignación aleatoria de nodos virtuales Es el sistema usado por last.fm en su sistema ketama**(author?)** [Jon07]. La asignación de nodos virtuales se realiza calculando el hash de cada nodo virtual, normalmente el nombre de su nodo padre seguido de algún tipo de numeración.

Figura 4.1.5: Distribución aleatoria de nodos virtuales

Al ser una distribución"aleatoria", el espacio de claves de cada nodo virtual puede variar, aunque no es un gran problema tal y como se ha descrito en el apartado anterior4.1.2. Este sistema presenta dos inconvenientes:

- Al añadir nuevos nodos al sistema, serán añadidos nuevos nodos virtuales al anillo. La redistribución posterior es muy costosa, ya que el espacio de claves de cada nodo queda parcialmente modificado y, por lo tanto, hay que realizar un análisis clave por clave para calcular la pertenencia de esta a cada uno de los nodos.
- Al aumentar el número de nodos virtuales de un nodo concreto, se modifica el espacio de claves de los nodos virtuales adyacentes, con la consecuente redistribución de claves.

Estos dos problemas no se presentan en un sistema orientado a caching como ketama. En el caso de añadirse un nuevo nodo, simplemente no se realiza una redistribución, sino que que vuelve a calcular el cache de las claves afectadas.

Asignación fija de nodos virtuales Es la estrategia seguida por Amazon en su sistema Dynamo[Giu07]. Se basa en la división del espacio de claves en un número fijo$Q$ de espacios, y asignarQ/N espacios a cada nodo, siendo$N$ el número de nodos. También se puede aplicar una distribución por pesos, asignando el número de espacios proporcionalmente al peso del nodo. Cuando un nodo abandona el sistema, los espacios libres dejados por este se reparten de manera uniforme entre los demás. Al introducirse un nuevo nodo al sistema, este roba espacios, también uniformemente, a los demás participantes.

Figura 4.1.6: Distribución fija de nodos virtuales

Este diseño solventa los problemas derivados de la distribución aleatoria. La distribución fija de nodos virtuales permite tener un control sobre los espacios de claves pertenecientes a cada partición. Al sustituir una partición completa de un nodo por otra, se distribuye el contenido integro, evitando calcular de un conjunto de claves cuales pertenecen a la nueva partición. Por ejemplo en4.1.6, si se añadiese un nuevo nodo, los nodos virtuales de este sustituirían diferentes nodos virtuales completamente, transfiriendo su contenido al nuevo nodo, sin cálculos previos.

4.1.3. Software de distribución.

Tweemproxy Tweemproxy es un proxy desarrollado por Twitter que permite distribuir las peticiones a Memcached y Redis utilizando diferentes algoritmos de distribución de claves. Además permite monitorizar las diferentes instancias de las bases de datos y excluirlas de la distribución en caso de fallo. Esto permite proveer a los sistemas de alta disponibilidad, en caso de ser servidores de cache, ya que nunca se realizaran peticiones a servidores caídos. Los diferentes algoritmos de distribución que implementa Tweemproxy, y entre los que se puede elegir son:

- Modula: distribuye las peticiones según hash(k) % num_servidores. Es el algoritmo que se ha explicado anteriormente4.1.2 y, como se ha visto, no provee elasticidad.
- Ketama: utiliza el algoritmo de hash consistente implementado por last.fm en libketama.
- Random: realiza una petición random a un servidor en cada interacción con este. No es un comportamiento deseable para el proyecto presente.

Tweemproxy ofrece varios algoritmos de hashing para calcular la distribución, entre ellos md5, crc16, crc32 y hsieh.

Analizando Tweemproxy encontramos dos problemas que impiden que este pueda ser usado en un sistema no cache, es decir, en un sistema que necesite mantener datos persistentes. Tweemproxy no permite añadir nuevas instancias de base de datos sin reiniciar el proxy, por lo tanto el sistema no será flexible. Además, Tweemproxy no redistribuye la carga de cada uno de los nodos en el caso de insertarse un nuevo nodo. Estos dos problemas impiden que el sistema mantenga persistencia, por ejemplo, al añadir nuevos nodos se perderá la referencia a esos datos al no haber sido migrados al nodo al que pertenecen.

4.2. Alta disponibilidad

La disponibilidad de un sistema software es el grado con el que los recursos del dicho sistema son accesibles por los usuarios. La disponibilidad no solo se refiere al tiempo durante el cual un sistema no está caído, sino que es el tiempo en el que el usuario puede interactuar con el sistema de manera correcta. Cualquier circunstancia que impida la correcta utilización de este por parte del usuario final se considerará como fallo del servicio y, por lo tanto, como tiempo no disponible del sistema. Estas circunstancias abarcan desde la caída completa del servicio hasta tiempos de respuesta demasiado altos[MS03].

También se entiende la alta disponibilidad en el ámbito de los datos. Un sistema es altamente disponible si se asegura la persistencia de los datos en cualquier circunstancia. Una perdida de datos puede ser crítica y con toda seguridad afecta a la continuidad de negocio. La caída de un servicio actualmente supone un serio problema.

4.2.1. Importancia de la alta disponibilidad

Desde hace pocos años, las empresas están sufriendo una intensa evolución hacia el modelo SaaS y el Cloud Computing1.1. La información de las empresas se encuentra distribuida a lo largo de muchos servidores que pueden encontrarse en diferentes lugares del mundo. Esta circunstancia, combinada con el hecho de que la mayor parte de la información de una empresa se encuentra en formato digital, plantea la necesidad de crear sistemas tolerantes a fallos y disponibles la mayor parte del tiempo posible. Una caída del servicio puede ser una catástrofe en sistemas críticos.

Podemos afirmar que no existe ningún sistema completamente fiable, existen infinidad de circunstancias que pueden hacer que un sistema falle. Algunos ejemplos son:

- Ataques terroristas. Como los ocurridos el 11 se Septiembre de 2001.

- Ataques a la seguridad informática. Virus, troyanos, etc.
- Desastres naturales como inundaciones, incendios, terremotos...
- Colapsos en las redes provocados por algún acontecimiento extraordinario.

Todos estas circunstancias son muy difíciles de prever y no se pueden evitar, pero se pueden proporcionar soluciones que permitan una cierta continuidad de negocio. Las empresas implicadas deben analizar estos riesgos y desarrollar sistemas altamente disponibles que garanticen un servicio lo más continuo posible.

4.2.2. Parámetros de la alta disponibilidad

El principal indicador de la alta disponibilidad es, por supuesto, el tiempo de disponibilidad del servicio. Prácticamente se puede asegurar que ningún sistema esté 100 % disponible a lo largo de un periodo de tiempo más o menos extenso. Para categorizar los sistemas en cuanto a su disponibilidad, existe el método de los "nueves"; el porcentaje de tiempo que un sistema puede ofrecer servicio al usuario. La tabla siguiente muestra las diferentes categorías en las que se puede emplazar un sistema:

Tiempo disponible	Tiempo no disponible	Tiempo caído al año	Tiempo caído a la semana
98 %	2 %	7.3 días	3 horas, 22 minutos
99 %	1 %	3.65 días	1 hora, 41 minutos
99.8 %	0.2 %	17 horas, 30 minutos	20 minutos, 10 segundos
99.9 %	0.1 %	8 horas, 45 minutos	10 minutos, 5 segundos
99.99 %	0.01 %	52.5 minutos	1 minuto
99.999 %	0.001 %	5.25 minutos	6 segundos
99.9999 %	0.0001 %	31.5 segundos	0.6 segundos

Cuadro 4.2.1: Categorías de los sistemas en cuanto a disponibilidad

Es importante señalar que no todos los sistemas necesitan cumplir una disponibilidad de 5 "nueves" o más. Cada sistema tendrá unas necesidades específicas y diferentes usos y usuarios. Por ejemplo, ciertos servicios, como un sistema de información bursátil en un empresa tendrá un uso muy bajo o nulo durante la madrugada, cuando ningún empleado se encuentra trabajando.

Esta categorización según el porcentaje de actividad del servicio se usa a menudo en las tareas de marketing, como medio para vender determinado servicio a usuarios u otras empresas. Este método es muy visual y muestra de una manera rápida la alta disponibilidad de un sistema. Sin embargo, el método de los "nueves" no tiene en cuenta muchos factores, que influyen en mayor medida a la disponibilidad del sistema de cara al usuario final.

- Los "nueves" son una media. En muchas ocasiones es más importante contar el tiempo máximo de recuperación del servicio que el tiempo de caída medio por semana.
- La medición sigue un modelo que no es real. Es muy difícil medir a priori la disponibilidad del sistema mediante herramientas genéricas. La disponibilidad real de un sistema se medirá en un caso real.
- No tienen en cuenta el sistema como algo global. Un sistema muy fiable y tolerante a fallos no sirve de nada si se encuentra detrás de un router averiado. La disponibilidad del sistema viene dada por todos los factores que influyen sobre este.

Resumiendo, el porcentaje de disponibilidad de un sistema puede calcularse mediante la siguiente fórmula:

$$A = \frac{MTBF}{MTBF + MTTR}$$

A es el grado de disponibilidad expresado en porcentaje, $MTBF$ es el tiempo medio entre fallos y $MTTR$ tiempo medio en reparar un fallo.

4.2.3. SPOF (Puntos simples de fallo)

La alta disponibilidad se basa principalmente en la replicación de todos los elementos del sistema. Un SPOF (Single Point Of Failure) es un elemento del sistema que no está replicado y que es susceptible a fallo. Para que un sistema sea HA no debe existir ningún SPOF. Cualquier elemento que intervenga en cierto proceso TI puede ser un punto de falla. Es importante no sólo replicar elementos de almacenamiento o servidores, sino también elementos de comunicación como routers. El fallo de un SPOF significa en la mayoría de ocasiones la caída completa del sistema. La replicación de todos los elementos de un sistema es algo muy costoso por lo que se ha de evaluar la probabilidad de fallo de cada elemento y el coste que produciría la perdida de este para lograr un equilibrio económico y de HA.

4.2.4. Alta disponibilidad en el almacenamiento

En la práctica, los discos duros son los componentes con más probabilidad de fallo. Cuantos más medios de almacenamiento estén involucrados en el sistema, mayor será la probabilidad de fallo de alguno de los discos. La probabilidad de fallo de un disco en un determinado tiempo sigue una distribución exponencial.

$$F(t) = 1 - e^{-\lambda t}$$

Donde:

- $F(t)$ es la probabilidad de fallo
- λ es la tasa de fallo expresada como 1/tiempo de fallo. Por ejemplo 1/300horas. Es la inversa de $MTBF$.
- t es el tiempo observado.

Así en un sistema con 100 discos, cada uno con un $MTBF$ de 800.000 horas, la probabilidad de que alguno de los discos falle a lo largo de un año será:

$$F(800.000) = 1 - e^{-\frac{8760*100}{800000}}$$
$$F(800.000) \approx 0.66$$

En un año, la probabilidad de fallo de algún disco será del 33 %. El tiempo estimado para que alguno de estos discos falle con una probabilidad del 90 % será:

$$\ln(1 - F(t)) = \ln(e^{-\lambda tn})$$
$$\ln(1 - F(t)) = -\lambda tn$$
$$\frac{\ln(1 - F(t))}{-\lambda n} = t$$
$$-\frac{\ln(1 - 0.9)}{\frac{100}{800000}} = t$$
$$2.3 * 8000 = 18400$$

18400 horas, aproximadamente 2 años en tener un 90 % de probabilidad de fallo en uno o más discos.

Por esta razón, la replicación de datos es uno de los aspectos más importantes de la alta disponibilidad. La perdida de recursos hardware es muy barata y se puede recuperar fácilmente. Los datos que se pierden son imposibles de recuperar si no se han salvado en otros discos de manera adecuada. En la mayoría de las empresas la información es el activo más importante por lo que una perdida o deterioro de esta significa un problema a nivel de continuidad de negocio.

Otro aspecto muy importante es la accesibilidad de los datos. La información crítica de una empresa debe estar continuamente accesible. Los datos que no son accesibles, no existen en la práctica.

RAID

RAID, acrónimo de Redundant Array of Independent Disks (conjunto redundante de discos independientes) es un estándar de almacenamiento que describe diferentes

maneras de combinar varios discos para proveer redundancia de datos**(author?)** [MS03]. Existen cinco niveles RAID que ofrecen diferentes niveles de redundancia, además de un nivel que no ofrece ninguna. Dependiendo de los diferentes niveles, RAID ofrece distintos beneficios como mayor integridad, mayor tolerancia a fallos, mayor rendimiento y mayor capacidad. Además de estos 6 niveles (5+1), existen aproximaciones resultantes de la combinación de varios, pudiendo obtener beneficios de ambos.

RAID-0

En este nivel no se ofrece redundancia de datos. Los datos que se escriben se dividen en segmentos iguales que se almacenan en cada uno de los discos del sistema RAID. El tiempo de escritura en cada disco es el mismo, por lo que la velocidad de escritura total es proporcional al número de discos. El tamaño ideal de un segmento de escritura es el tamaño de los datos a escribir dividido entre el número de discos disponibles.

La disponibilidad de los datos en este nivel no sólo no está asegurada, sino que disminuye a medida que aumenta el número de discos. Como la información está dividida entre todos los discos y no existe redundancia, un fallo en un disco produce la imposibilidad de obtener ninguna información. La probabilidad de que un disco falle aumenta proporcionalmente al número de discos del sistema, según se ha visto en esta sección.

RAID-1

En un sistema RAID1, cada byte que se escribe en un disco se escribe en uno o más discos iguales. La sincronización es (casi) simultanea. Si existe un fallo en uno de los discos el segundo continuaría operando con normalidad. Un sistema RAID1 no sigue una replicación maestro/esclavo; en esta aproximación no existe ningún maestro, las escrituras son simultaneas en todos los discos y todos ellos pertenecen al mismo nivel jerárquico.

El rendimiento en las lecturas mejora ligeramente, ya que el sistema lee el mismo dato de todos los discos y selecciona el que ha tardado menos en responder. La mayor ventaja de RAID1 es la protección de los datos cuando alguno de estos falla. Si uno de los discos falla se continuará operando con los discos restantes sin que existe una interrupción de servicio.

Las desventajas de RAID1 son varias. La primera es el gasto de recursos. Toda la información de un disco esta replicada en los demás, por lo que los datos ocupan un 100, 200, 300... más. Esto supone un “malgasto” de recursos muy importante. La otra desventaja es el retraso en las escrituras. Al contrario de lo que ocurre el las lecturas, una operación de escritura será tan rápida como el disco más lento del sistema.

RAID-3, 4, y 5

Estos niveles utilizan diferentes tipos de paridad. En estos sistemas no se necesita mantener una copia completa del disco. El sistema requiere un espacio extra en cada disco para guardar los valores de paridad de datos del sistema completo. Estos valores se calculan aplicando operaciones lógicas (XOR, OR) a los bloques de datos de cada disco. En general, los sistemas RAID-3, 4 y 5 requieren de un espacio adicional de 20 al 25 por ciento, dependiendo del número de discos del sistema.

En caso de fallo de uno de los discos, los datos se calculan a partir de los valores de paridad de los demás discos. En caso de fallar más de un disco, los datos no se podrán reconstruir y será necesario recuperarlos desde un disco de backup.

Estos sistemas tienen un rendimiento bastante peor que los otros en lo referente a el tiempo de escritura. Por cada bloque de datos que se escribe, se necesitan actualizar los valores del disco de paridad. Esto significa leer los datos de los demás discos, realizar las operaciones lógicas correspondientes, y almacenar estos valores en el disco de paridad.

Las lecturas se realizan sin calcular valores de paridad por lo que su rendimiento es el mismo que un disco no RAID.

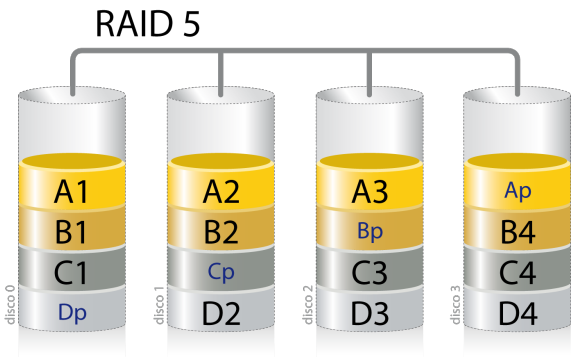

Figura 4.2.1: Distribución de datos en un sistema RAID-5

4.2.5. SAN y NAS, redes de almacenamiento

Un SAN es una red de almacenamiento comunicada con el resto del sistema a través de redes de fibra de alta velocidad. NAS es también una red de almacenamiento pero no se conecta a resto del sistema mediante redes de alta velocidad, sino a través de una red de área local común. Los sistemas NAS son mucho más económicos que los SAN ya que no requieren de una red de alta velocidad dedicada. En contraposición, los sistemas SAN son mucho más rápidos y permite almacenar cantidades de datos casi ilimitadas.

Las ventajas de SAN son las siguientes:

- Almacenamiento compartido. Permite que varios servidores sean conectados a un mismo grupo de discos.

- Alta disponibilidad. La posibilidad de poder conectar varios servidores a los mismos conjuntos de datos permite replicar recursos en cuanto a servidores.
- Mejora de rendimiento en el acceso a los datos. Las redes de fibra óptica permiten transmitir datos a gran velocidad. Actualmente alcanzan velocidades de 4 gigabits por segundo.
- Transmisión a grandes distancias. Otra ventaja de las redes de fibra óptica es la distancia a la que pueden transmitir los datos sin necesidad de routers intermedios. Una red SAN puede transmitir datos hasta distancias de 10Km.

Canal de fibra

Canal de fibra o Fibre Channel (FC) es una tecnología de transmisión de datos a alta velocidad entre mainframes y dispositivos de almacenamiento. Se trata de una interfaz de transferencia de datos que soporta diferentes protocolos de transporte como SCSI o IP. Este soporte multiprotocolo permite reunir bajo una misma tecnología de interconexión las funcionalidades de las redes (networking) y las de E/S de alta velocidad (principalmente memorias de masa). A pesar de su nombre, FC puede operar tanto sobre cables de fibra óptica como de cobre a distancias hasta 10km sin uso de repetidores. Las comunicaciones a través de un canal FC son directas punto a punto, no existe direccionamiento como en una red. Por esta razón, la sobrecarga de datos en la transmisión es mínima y en la trasmisión solo interviene el hardware**?**].

Un canal de fibra consta de dos fibras unidireccionales que permiten una conexión full-duplex. Ambas fibras estás conectadas a un puerto transmisor (Tx) y un puerto receptor(Rx) en cada extremo. Existen diferentes topologías de conexión, dependiendo de las conexiones entre los diferentes elementos.

Punto a punto

En esta topología solo existen dos elementos que se interconectan entre sí. Un N_Port asigna automáticamente la dirección al otro N_Port. La expansión es posible añadiendo más N_Ports a los Nodos.

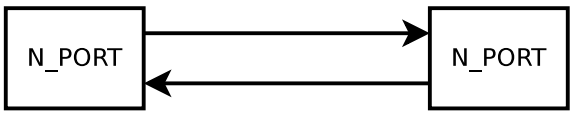

Figura 4.2.2: Topología punto a punto en FC.

Arbitrated Loop (FC-AL)

El transmisor de un puerto de conecta al siguiente, sucesivamente hasta formar un anillo. Los datos se transmiten a través de los diferentes nodos hasta alcanzar el nodo

destino. Dentro del Loop, cada Port se identifica por una dirección de 8 bits (AL_PA o Physical Address) que se asigna automáticamente durante la inicialización del Loop.

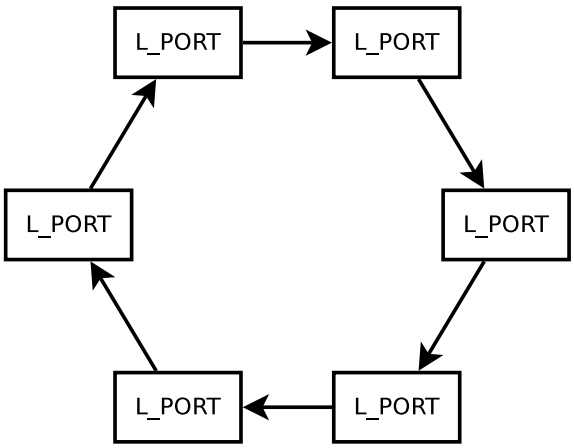

Figura 4.2.3: Topología de bucle arbitrado de FC

Conmutada (Fabric)

La conexión entre los nodos se realiza a través de un conmutador. Este conmutador (fabric), dirige el tráfico desde el nodo origen hasta el nodo destino.

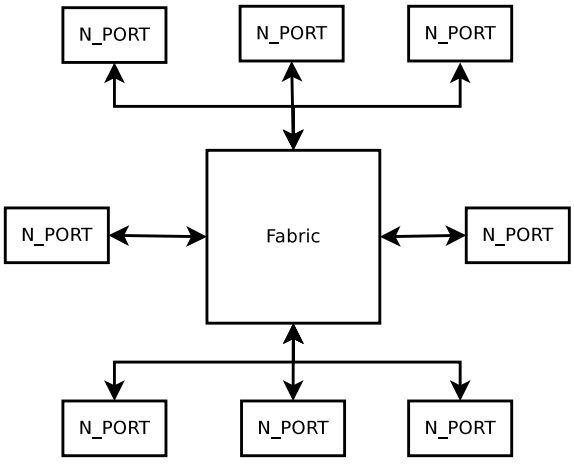

Figura 4.2.4: Topología conmutada de FC

4.2.6. Redis Sentinel

Redis Sentinel es la solución de Redis para proveer de alta disponibilidad a un conjunto de instancias. Este sistema formará parte del cluster Redis, actualmente en producción2.2.5.

Redis Sentinel es un conjunto de monitores que observan el estado de las instancias Redis, notifican en caso de comportamiento anómalo, y pueden iniciar procesos de failover[1]. Las diferentes instancias Sentinel pueden distribuirse a lo largo del sistema para evitar puntos simples de fallo. Redis Sentinel se combina con el sistema de replicación Maestro/Esclavo para, además de mantener una redundancia de datos, poder asegurar una continuidad de servicio en caso de fallo.

Cada Sentinel podrá monitorizar diferentes instancias Redis, que deberán ser instancias maestras de un determinado cluster. Estos comprobarán el estado de cada nodo Redis y en caso de fallo iniciarán un proceso por el cual un esclavo del nodo defectuoso se convertirá en maestro.

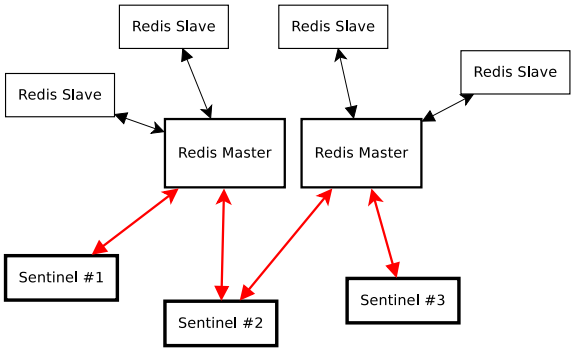

Figura 4.2.5: Ejemplo de despliegue con monitores Sentinel compartidos.

Comunicación

Como se ha mencionado, cada Sentinel puede monitorizar varias instancias Redis, y cada nodo Redis puede ser monitorizado por varios Sentinels. Para llegar a un acuerdo sobre el estado de determinada instancia los Sentinels deberán comunicarse entre sí, decidiendo si esta está realmente fallando o es un error de "percepción" de un Sentinel en concreto. Esta comunicación requiere del conocimiento de un Sentinel por parte de los demás.

En este sistema, el descubrimiento de nuevos Sentinels es automático, y se realiza a través de mensajes Pub/Sub. Cada Sentinel enviará "saludos" a un canal compartido en el maestro monitorizado. Cada Sentinel estará subscrito a este canal, de manera que cada nuevo nodo notificará al resto de su incorporación al sistema. El "saludo" que enviará cada Sentinel contendrá información sobre su dirección IP, su runid y la posibilidad se que pudiera iniciar un proceso de failover. Así, en caso de fallo los Sentinels se comunicarán externamente al canal, ya que este no existirá en dicho caso.

[1]Configuración de equipos en la que un segundo equipo de hace cargo de las funciones del principal en caso de detención de éste.

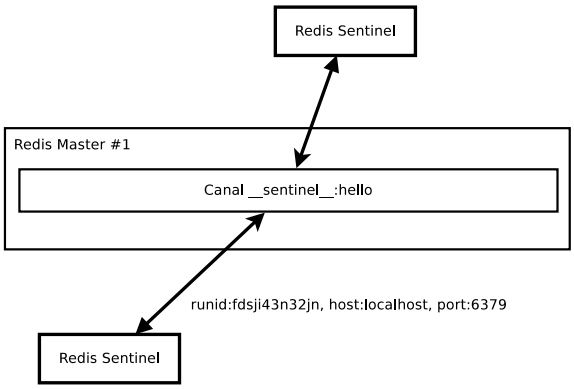

Figura 4.2.6: Ejemplo de descubrimiento por medio de publicación en canal compartido.

Estado de los maestros

Cada Sentinel comprobará el estado de los maestros monitorizados mediante el envío de mensajes PING a intervalos determinados. Los maestros que estén funcionando correctamente responderán a dicho mensaje. En el momento que no se reciba ninguna respuesta por parte de la instancia Redis en un tiempo acordado, el Sentinel marcará a esta como SDOWN (Subjetivamente caído).

Existen dos estados de fallo: SDOWN y ODOWN. El primero se alcanzará cuando se produzca la condición explicada en el parrafo anterior. Este estado es subjetivo, ya que no hay pruebas suficientes de que realmente sea la instancia la que este fallando, y no el Sentinel. Para que un nodo se considere fallando objetivamente (ODOWN) debe existir una cantidad prefijada de Sentinels que lo marquen como SDOWN. Este quórum es fijado previamente y suele ser 2 o 3, para garantizar fiabilidad. Cuando el quórum es alcanzado, el nodo pasará al estado ODOWN y se realizarán las acciones convenientes.

Proceso de failover

En el momento que un nodo pase al estado ODOWN se puede iniciar un proceso de failover para mantener una continuidad de servicio. El objetivo de este proceso es elegir y promocionar un esclavo del maestro que este fallando para poder seguir realizando peticiones a este. Este proceso sigue una serie de pasos que se detallan a continuación:

1. Se selecciona el líder del proceso. Los demás Sentinels serán observadores.
2. El líder del proceso, mediante determinadas reglas, seleccionará que esclavo debe promocionar.
3. El líder promocionará el esclavo seleccionado mediante el comando “SLAVEOF NO ONE”.

4. Los demás nodos (esclavos del nodo que falla) serán reconfigurados para ser esclavos del nuevo maestro.

5. Cada Sentinel, tanto el líder como los observadores, se reconfigurará para tomar como maestro al esclavo promocionado.

Todo este proceso será monitorizado por todos los Sentinels, y estos notificarán mediante Pub/Sub de todas las acciones que se están llevando a cabo en cada momento.

Capítulo 5

Desarrollo

En el capítulo de Software2, se ha documentado el comportamiento de los sistemas de Rush y PopBox, así como su arquitectura interna. Como se ha visto, PopBox es un sistema de buzones de alto rendimiento, y Rush es un proxy asíncrono.

A la hora de realizar un diseño e implementarlo, se ha decidido PopBox como mejor opción. La primera razón de está decisión es la necesidad de persistencia del sistema PopBox. PopBox es un sistema de buzones, por lo tanto necesita que los datos almacenados en estos sean persistentes y estén disponibles el mayor tiempo posible.

El desarrollo de un middleware de alta disponibilidad y escalabilidad ha supuesto un análisis profundo de todos los sistemas de alta disponibilidad implantados actualmente. El propósito del sistema planteaba la necesidad de mantener ciertos aspectos del desarrollo original, como la escalabilidad a nivel de Agentes y el rendimiento.

El diseño original de PopBox proveía una solución de escalabilidad basada en módulo, por lo tanto no permitía añadir nuevos nodos REDIS sin tener que realizar una completa redistribución de todas sus claves, algo verdaderamente costoso en entornos con mucha carga. Tampoco proveía una configuración consistente entre los distintos agentes, lo que podía derivar en distintas distribuciones de claves dependiendo del agente al que se realizase la petición, es decir, inconsistencia de datos. En cuanto a la alta disponibilidad de PopBox, el sistema seguía una redundancia basada en maestro/esclavo sin ninguna detección de failovers y muy poca automatización. En un sistema con un número muy alto de instancias PopBox e instancias Redis, la estrategia seguida era inmanejable, además de dejar sin servicio al sistema durante un tiempo considerable.

Durante el desarrollo del proyecto se diseñaron diferentes soluciones, siguiendo un proceso iterativo. Estos diferentes diseños han dado lugar a tres aproximaciones distintas. Para mantener todos estos diseños e implementaciones se ha utilizado la herramienta de control de versiones Git y la forja de desarrollo GitHub. Este proyecto es un fork del repositorio principal de PopBox en Telefónica I+D. La dirección web del repositorio del presente proyecto es `https://github.com/fgodino/PopBox`

5.1. Primera aproximación

La primera aproximación fue diseñar un proxy que actuase de intermediario entre los diferentes agentes y las diferentes instancias de Redis. El diseño de la arquitectura de deploy consiste de 1..n proxies conectados entre los Agentes y Redis. Los diferentes Agentes se comunican con los proxies utilizando el protocolo Redis, y los proxies se comunican con Redis utilizando este mismo protocolo.

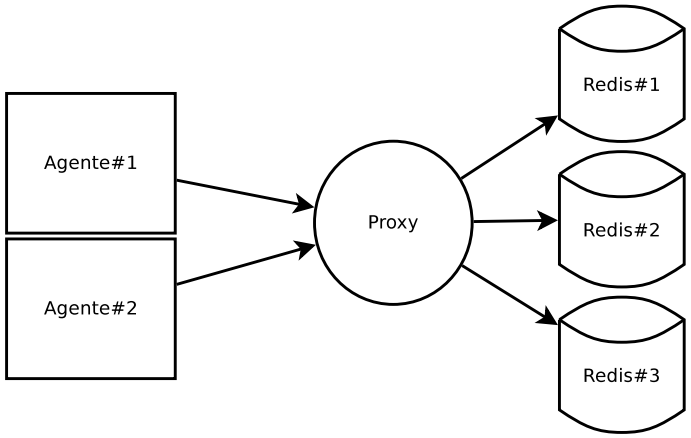

Figura 5.1.1: Arquitectura de deploy en la primera aproximación para escalabilidad

5.1.1. Arquitectura

El comportamiento del proxy es muy sencillo. El proxy actúa de intermediario entre los diferentes agentes y las instancias REDIS. El proxy es completamente transparente al Agente, es decir, el agente realiza una petición al proxy de la misma manera que lo haría con una instancia REDIS, utilizando el mismo protocolo y las mismas llamadas. Si el proxy fuese eliminado de la arquitectura de deploy, PopBox se comportaría de la misma manera haciendo peticiones a un REDIS. Los pasos que realiza el proxy para encaminar una petición del agente a un servidor Redis son los siguientes:

1. Proxy recibe una petición REDIS.
2. Proxy descompone la petición en los campos *orden, clave* y *cuerpo:* hay que tener en cuenta que una petición REDIS es en la mayoría de los casos de la forma *[comando clave [campos]]*, por ejemplo [HGET clave campo1].
3. Proxy calcula, mediante el algoritmo de distribución, la base de datos destino.
4. Proxy crea un *stream pipe* entre la salida de la instancia redis seleccionada REDIS y la entrada de PopBox mediante Sockets Stream (TCP). Esto permitirá que la respuesta de REDIS sea dirigida directamente a PopBox, sin pasar por Proxy y disminuyendo el throughput.

5. Proxy realiza la petición recibida a la instancia REDIS calculada en el paso 3.

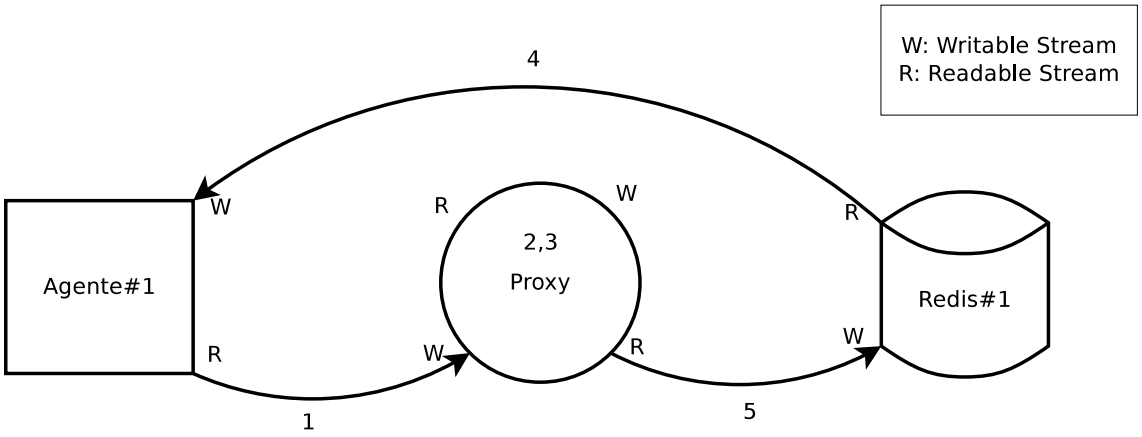

Figura 5.1.2: Diagrama de pasos realizados para dirigir una petición Redis a su correspondiente instancia.

El comportamiento anterior tiene algunas excepciones. Algunos comandos REDIS no reciben ninguna clave, ya que son consultas referidas a la base de datos en sí. Estos comandos comprueban que exista conexión con la Base de Datos, eliminan todos los datos y realizan operaciones maestro/esclavo, entre otros. Como es obvio, el cálculo de la instancia REDIS a la que irá dirigido la petición basado en la clave de la petición no se podrá realizar, al no existir dicha clave. Este escenario es poco usual en el sistema PopBox, ya que la mayoría de peticiones son dirigidas a inserción o extracción de datos. Para evitar errores en el parseo de la petición, el proxy comprobará que el comando recibido desde PopBox esté contenido en el conjunto de comandos que requieran clave. Si no está contenido, el proxy reenviará la petición recibida a cada una de las instancias REDIS dentro del sistema. PopBox será respondido con un objeto(multibulk) conteniendo cada una de las respuesta de cada REDIS implicado. El Agente, por lo tanto, deberá ser consciente de que para este tipo de peticiones la respuesta recibida no será la habitual, sino que recibirá un conjunto de respuestas.

5.1.2. Algoritmo de distribución

Para la distribución de claves en las distintas instancias REDIS se diseño un algoritmo de hash consistente basado en libketama y Dynamo. Estas dos soluciones proveen un algoritmo de distribución de claves utilizando el concepto de “anillo de hash”, mencionado en4.1.2, pero no proponen ninguna solución para la redistribución y el bootstrapping en caso de añadirse o eliminarse instancias DB del anillo.

El algoritmo que se siguió en esta aproximación está basado en la distribución aleatoria de nodos virtuales dentro de un anillo hash. Como he mencionado anteriormente4.1.2, esta estrategia permite distribuir la carga de los servidores de manera

proporcional a la capacidad de cada una. Cada nodo dentro del sistema es un objeto que posee como clave el nombre de la base de datos; y como valores, su host, su puerto y su peso. Cada uno de estos nodos tendrá un número variable de nodos virtuales distribuidos a lo largo de un anillo de hashing. La inserción de un nuevo nodo en el sistema sigue el siguiente algoritmo.

Algorithm 5.1 Inserción de un nodo en el sistema utilizando distribución aleatoria de nodos virtuales

Require: continuum = objeto que contiene el par clave-valor de la forma keyRing : nodo, ring = array de hashes de los nodos virtuales, n = número de nodos virtuales, node = nombre del nuevo nodo

1: **for** $i = 1 \rightarrow$n **do**
2: vNode = node + : + i
3: insertar hash(vNode) en keyRing
4: insertar en continuum el par node : hash(vNode)
5: **end for**
6: ordenar keyRing //Importante para que sea un anillo, los valores deben ser ordenados.

El algoritmo anterior simplemente realiza el hash del nodo virtual, que es la concatenación del nombre del nodo padre más un valor creciente en el espacio $[0, n)$, separados por dos puntos (:). El algoritmo de hash MD5 garantiza una distribución bastante uniforme de cada uno de estos nodos virtuales, y su homogeneidad será proporcional a la cantidad de nodos insertados.

El cálculo del nodo al que pertenecería cierta clave se realiza calculando el espacio de claves al que pertenece el hash de dicha clave. La distribución por nodos virtuales permite que el tamaño total de la suma de los espacios de claves de cada nodo virtual sea proporcional al peso de cada nodo perteneciente al sistema. El algoritmo realiza una búsqueda dicotómica de la clave, con el fin de encontrar el valor superior más próximo al hash de la clave dentro del anillo de hashing. Para ello, el vector que implementa este anillo debe estar ordenado y sus extremos deben estar conectados; es decir, el siguiente valor más próximo a la ultima posición del vector es el valor situado en la primera posición del vector. El cálculo del nodo al que pertenece cierta clave es el siguiente.

Algorithm 5.2 Cálculo del nodo al que pertenece una determinada clave

Require: continuum = objeto que contiene el par clave-valor de la forma keyRing : nodo, ring = array de hashes de los nodos virtuales, n = número de nodos virtuales, clave = nombre de la clave a calcular.

1: claveHash = hash(clave)
2: keySpace = DICOTOMIC(keyRing, claveHash) // busqueda dicotómica de claveHash dentro de keyRing
3: node = GET(continuum, hashSpace) // obtiene el nodo asociado a ese keySpace
4: **return** node

La búsqueda dicotómica dentro de un vector tiene un complejidad $O(\log n)$, por lo que el cálculo del espacio de claves asociado a determinada clave es poco costoso.

El borrado de un nodo tiene un funcionamiento similar a la inserción. Para cada nodo virtual de la forma $nodoPadre : i$ donde i es un número en el intervalo $[0, n)$ (n número de nodos virtuales), se calcula su hash correspondiente. Cada hash calculado se borrará del anillo de hashing y también del objeto clave-valor que asocia cada espacio de claves con su nodo.

5.1.3. Redistribución

Como he mencionado antes, las soluciones de escalabilidad propuestas por Dynamo, ketama, Voldemort o Tweemproxy, no proponen estrategias para la redistribución de las claves de cada DB en caso de inserción o borrado de nodos, o de un re-arranque del sistema en caso de fallo. Siguiendo el algoritmo de hashing consistente se consigue minimizar de manera considerable el número de claves que se han de redistribuir. En el caso de PopBox, un sistema que necesita persistencia en sus bases de datos, estas claves han de ser redistribuidas.

La inserción de un nuevo nodo requiere que claves de las distintas bases de datos que ya estaban en el sistema migren hacia el nuevo nodo insertado. En un anillo de hashing sin nodos virtuales esta migración solo se realizaría desde una única base de datos, ya que el espacio de claves que se modificaría pertenecería a un único nodo. En el caso de una implementación con nodos virtuales, los espacios de claves que se modifican pueden pertenecer a distintos nodos padre, por lo que la migración de las claves ha de hacerse desde varias bases de datos a la nueva base de datos insertada en el sistema.

Para conocer que claves se han de migrar y cuales no, el sistema deberá calcular, una por una, que claves pertenecen al espacio de claves del nuevo nodo introducido. El algoritmo que calcula la pertenencia o no de las claves al nuevo nodo es el siguiente:

Algorithm 5.3 Redistribución de claves en la inserción de un nuevo nodo.

Require: nodos = bases de datos del sistema, nuevoNodo = nodo a insertar en el sistema

```
1: INSERTNODE(nuevoNodo)
2: for all nodo in nodos do
3:    claves = GETKEYS(nodo)
4:    for all clave in claves do
5:       nodoDestino = GETNODE(clave)
6:       if nodoDestino = nuevoNodo then
7:          MIGRAR(clave, nuevoNodo);
8:       end if
9:    end for
10: end for
```

Este mismo algoritmo se llevará a cabo en el arranque del programa, con cada uno de los nodos que se instancien durante el mismo. En un arranque sucio (con datos en los diferentes REDIS), esto puede llevar algún tiempo, por la cantidad de datos que se han de migrar entre bases de datos.

Algo que no se menciona en este algoritmo es la posibilidad de que existan claves ya insertadas en el nuevo nodo que se va a introducir en el sistema. Puede darse el caso de que, en un rearranque, no se instancie una base de datos que contiene claves. Al añadir dicha base de datos al sistema, esas claves no serán accesibles en algunos casos ya que el algoritmo de distribución espera encontrarlas en otra base de datos. Todas las claves estarán en el sistema, pero no serán accesibles ya que no se habrá realizado una redistribución de estas. Se propone el siguiente escenario.

1. Un despliegue PopBox está provisto de dos bases de datos: Redis#1 y Redis#2.
2. Un usuario empieza ha insertar transacciones en PopBox, almacenándose diferentes claves en Redis#1 y Redis#2.
3. El despliegue de PopBox se interrumpe.
4. Se vuelve a arrancar PopBox, pero sustituyendo Redis#2 por Redis#3.
5. Al cabo de un tiempo se vuelve a introducir Redis#2.

Al insertar Redis#2 por segunda vez, muchas de las clave contenidas en este deberían estar contenidas en Redis#3. Por lo tanto, el anillo de hashing mapeará estas claves como contenidas en Redis#3, y no las encontrará. Para solventar este problema, habrá que comprobar si las claves contenidas en un nodo nuevo deberían pertenecer a ese nodo teniendo en cuenta la nueva configuración. La solución que he llevado a cabo es realizar, cada vez que se añade un nodo, la redistribución que se efectúa cuando se elimina un nodo del sistema, que veremos a continuación.

La eliminación de un nodo del sistema, requiere que las claves contenidas en este se distribuyan entre los nodos restantes del sistema. Para ello, se deberá calcular a que nodo pertenecerá cada una de sus claves según la nueva configuración. La operación de redistribución de borrado no es muy costosa, ya que solo obtiene todas las claves de una única base de datos. El algoritmo que sigue esta operación es el siguiente.

Algorithm 5.4 Redistribución de claves en la inserción de un nuevo nodo.

Require: borraNodo = nodo a eliminar sistema

```
1: REMOVENODE(borraNodo)
2: claves = GETKEYS(borraNodo)
3: for all clave in claves do
4:     nodoDestino = GETNODE(clave)
5:     MIGRAR(clave, nuevoNodo);
6: end for
```

5.1.4. Alta disponibilidad

Para dotar al sistema de alta disponibilidad se optó por la redundancia de datos mediante el mecanismo de maestro/esclavo. Mediante este mecanismo en Redis, el maestro replica cada dato en los demás esclavos. En el modelo diseñado, el maestro tiene un solo nodo esclavo, y un esclavo puede ser maestro de otro a su vez. De esta manera se genera una cadena de replicación que dispersa la carga de la replicación entre el maestro y los esclavos.

A su vez, para evitar puntos únicos de fallo, un despliegue estará provisto de al menos un Proxy Maestro con una cadena de Proxies esclavos. Cada Proxy estará conectado a una serie de instancias redis, de manera que cada instancia redis esté conectada a un solo Proxy, que será el encargado de monitorizar el comportamiento de esta.

Un balanceador de carga se encargará de repartir la carga entre los Proxies maestros y detectar el fallo de un agente, para delegar todas las peticiones a su esclavo. El balanceador contendrá, por cada proxy maestro, una cola con todos sus proxies esclavos de manera que si falla este, se dirijan las peticiones al siguiente en la cola.

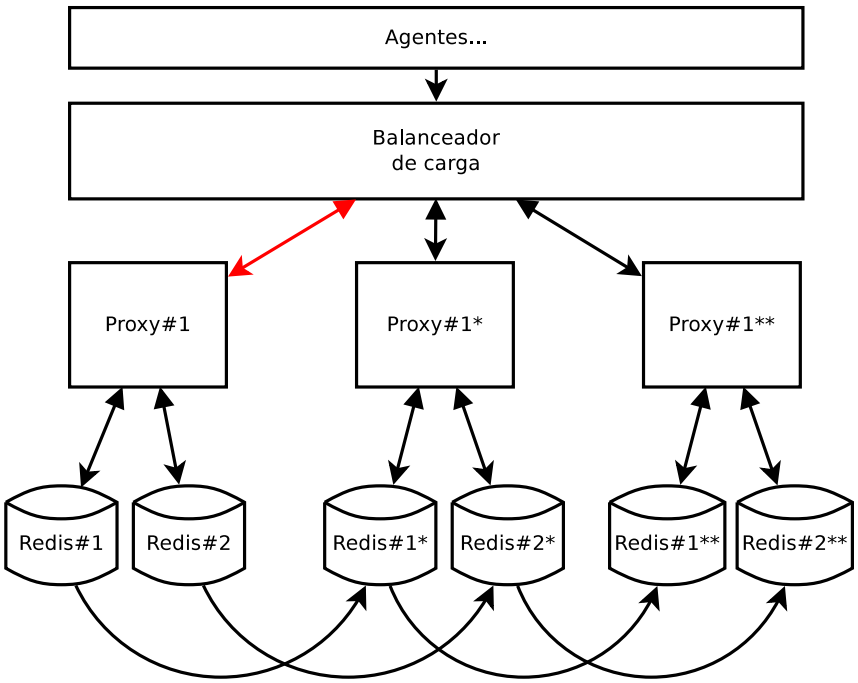

Figura 5.1.3: Arquitectura de Alta disponibilidad en la primera aproximación

El proxy dirigirá todas las peticiones redis a la correspondiente instancia redis según el valor de la clave a insertar, como se ha visto en el apartado de distribución5.1.2. Este proxy envía paquetes keep-alive a cada instancia cada cierto tiempo. En el momento que se detecta un fallo en una de las instancias redis controladas por el proxy maestro se inicia un mecanismo de failover que realizará las siguientes acciones:

1. Promocionar cada uno de las instancias redis esclavas de las actuales(las maestras).

2. El proxy comenzará a contestar a todas las peticiones recibidas con un mensaje de error según el protocolo Redis.

3. El balanceador de carga comenzará a dirigir las peticiones al siguiente proxy en la cola.

En el momento de agregar un nuevo proxy con sus correspondientes instancias Redis, o al volver a agregar un proxy que ha fallado, dicho proxy se agregará al final de la cola de proxies, y sus instancias Redis serán esclavas de las instancias Redis del último proxy. En despliegues con una cadena de maestros-esclavos de longitud $\geqq 3$, la replicación de todos los datos al nuevo esclavo la llevan a cabo instancias redis que no están recibiendo peticiones a parte de aquellas de sincronización, por lo que la carga en el maestro se ve disminuida y por consiguiente su tiempo de respuesta.

En primer lugar se optó por realizar las comunicaciones con el proxy mediante el protocolo TELNET[1], pero por cuestiones de simplicidad y por mantener una unicidad en las comunicaciones con PopBox, se diseño una interfaz HTTP para realizar las operaciones de ADD, DELETE y GET.

5.1.5. Problemas

Durante el diseño e implementación de esta solución se han encontrado varios problemas que, o bien convierten al sistema en poco escalable, o merman su rendimiento en un sistema en producción.

El primer problema es el cuello de botella que se produce en cada uno de los proxies. Cada proxy actúa como una sola instancia Redis, repartiendo las peticiones a las verdaderas instancias Redis detrás de este. La escalabilidad que proporcionaba el tener varias instancias Redis y varias peticiones a distintos Redis ejecutándose a la vez, se ve mermada por la necesidad de encolar cada petición y cada respuesta entre los Agentes y los nodos Redis. En la implementación original, cada llamada a una instancia Redis se realizaba de manera asíncrona, distribuyéndose la carga entre las distintas bases de datos disponibles. Por lo tanto, un proxy es irremediablemente un cuello de botella.

El segundo problema está referido a la escalabilidad. Como he explicado antes5.3.2, la inserción o borrado de un nodo Redis supone la redistribución de algunas claves contenidas en el sistema. El algoritmo de redistribución extrae todas las claves de cada nodo Redis, las analiza una por una y, en caso de pertenecer a un nodo distinto del actual, las migra. La extracción de todas las claves del sistema es un proceso muy costoso. La implementación de esta aproximación utiliza un patrón para extraer todas las claves. El comando Redis utilizado para extraer de cierto nodo todas las

[1]TELecommunication NETwork

claves que sigan determinado patrón es *KEYS pattern,* donde pattern es el patrón de búsqueda. Así pues, el comando para extraer TODAS las claves de un nodo Redis es *KEYS *.* Redis desaconseja el uso de este comando en entornos de producción, tal y como menciona en la documentación[San]: "*Warning: consider KEYS as a command that should only be used in production environments with extreme care. It may ruin performance when it is executed against large databases. This command is intended for debugging and special operations, such as changing your keyspace layout. Don't use KEYS in your regular application code.*"

5.1.6. Conclusión

Este diseño es muy fiable ya que no involucra intercambio de mensajes entre los distintos nodos y cada modulo está completamente desacoplado de los demás. Sin embargo, existe un problema muy importante en la escalabilidad del sistema y en el rendimiento del mismo. Dados los problemas anteriores, se concluyó que este diseño no es apto para sistemas en producción.

5.2. Segunda aproximación

En esta segunda aproximación se optó por crear un servidor de configuración que distribuyese la configuración de distribución, es decir, el anillo de hashing, entre los diferentes Agentes del sistema. Esta solución evita el cuello de botella generado por el proxy en la aproximación anterior. Este diseño utiliza un sistema Pub/Sub[2] ofrecido por Redis para la comunicación y el descubrimiento de nuevos servidores de configuración y nuevos agentes.

5.2.1. Arquitectura

La arquitectura de este diseño consta de varios servidores de configuración y varios agentes que intercambian mensajes a través de una instancia Redis. Además, los Agentes comunican directamente con las instancias Redis encargadas del almacenamiento de transacciones y colas.

[2]Publish/Subscribe: sistema de editores/subscriptores.

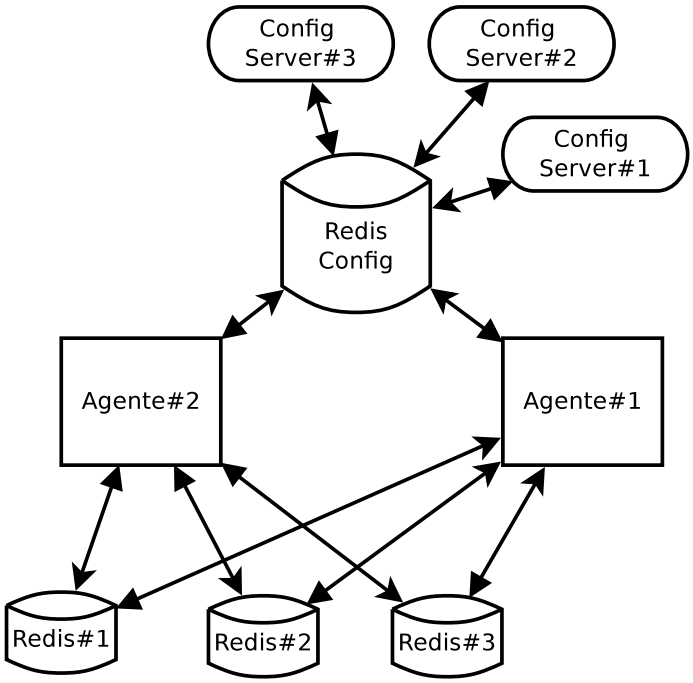

Figura 5.2.1: Arquitectura de PopBox en la segunda aproximación

Cada uno de los servidores de configuración se encargará de calcular el anillo de hashing y de distribuirlo entre los Agentes y los demás servidores de configuración. Es importante el mantener varias instancias del servidor de comunicación para evita puntos únicos de fallo y dotar al sistema de Alta Disponibilidad. Los servidores contienen mecanismos para garantizar la fiabilidad y evitar inconsistencias en los datos. Cualquier comunicación entre los nodos se llevará a cabo mediante la instancia Redis de configuración.

El nodo de configuración Redis almacenará la configuración del anillo de hashing, guardando la configuración (host, puerto) de cada una de las instancias Redis de almacenamiento, y la configuración del anillo de hashing, necesaria para calcular la distribución de claves. Así mismo permitirá la comunicación entre los Agentes y los servidores de comunicación, y permitirá a ambos descubrirse entre sí, sin necesidad de existir una configuración previa.

Los Agentes enviarán directamente las peticiones de almacenamiento y extracción a la instancia Redis pertinente, dependiendo de la clave que se almacene. Este cálculo se llevará a cabo mediante un algoritmo de hash consistente5.1.2. El anillo de hash necesario para calcular la distribución de claves se encontrará en el nodo de configuración Redis, por lo que cada Agente tendrá que sincronizar esta información mediante mecanismos que explicaré más adelante.

5.2.2. Descubrimiento

Todos los Agentes y los servidores conocen la existencia de los otros sin necesidad de conocer "a priori" ningún dato sobre estos. El descubrimiento de Agentes se realiza mediante Pub/Sub, por lo tanto cada Agente solo necesita conocer la dirección del canal de subscripción para ser reconocido por el sistema. Para ello, cada Agente o servidor de configuración publicará un saludo en un canal determinado. Los canales existentes son "agent:new" y "migrator:new".

Un agente nuevo publica un mensaje cada 3 segundos en el canal "agent:new". El mensaje contendrá un identificador único de instancia, de manera que si se aborta y se vuelve a arrancar ese mismo agente tendrá un identificador único. En el momento en el que cada servidor de configuración es notificado de un nuevo agente, se pone en marcha un sistema que monitoriza dicho agente. El sistema comprueba que el agente envíe mensajes 'keep-alive' cada 3 segundos. Si en 9 segundos no se ha recibido ningún mensaje, ese agente se marcará como caído, y no se tendrá en cuenta a la hora de realizar sincronizaciones.

Los servidores de configuración funcionan de manera similar. En este caso, cada servidor de configuración es independiente de los demás, no necesita del conocimiento de los demás servidores para funcionar correctamente, pero se utiliza este mecanismo para evitar que dos servidores modifiquen el anillo de hashing en un mismo instante de tiempo. Además, en el momento de realizarse una sincronización exitosa, cada servidor de configuración recibirá una copia idéntica del anillo de hashing.

5.2.3. Distribución y redistribución

El mecanismo de distribución y el se redistribución en esta segunda aproximación es el de anillo de hashing con distribución aleatoria de nodos virtuales. Véase 5.1.2 y 5.3.2.

5.2.4. Sincronización

La sincronización es el proceso por el cual cada cliente obtiene una copia del anillo de hashing, necesario para calcular la distribución de las claves en las diferentes instancias Redis. La sincronización requiere que todos los agentes activos reciban una copia de la nueva distribución antes de comenzar a recibir peticiones. En caso de que algún agente empiece a operar con una configuración de distribución obsoleta y diferente de los demás agentes se produce una inconsistencia de datos. Por ejemplo, en un sistema con un servidor de configuración SC#1, y tres agentes A#1, A#2, y A#3; puede darse el siguiente caso:

1. El servidor de configuración SC#1 inicia una sincronización, iniciada por la inserción de una nueva instancia Redis.
2. Los agentes A#1 y A#2 reciben el cambio.

3. El agente A#3 recibe una petición e inserta la clave generada en una de las instancias Redis.

4. El agente A#1 recibe una petición Pop de la clave insertada en el paso anterior y recibe una respuesta vacía.

Esto puede ocurrir debido a que el mapeo de las claves en los agentes A#1 y A#2 es diferente del mapeo de claves en el agente A#3.

Para que esto no ocurra debe existir un mecanismo de comunicación entre todos los agentes y el servidor de configuración que inicie una sincronización. Para que el servidor de configuración realice un cambio en el anillo de hashing, todos los agentes deben notificar su conformidad. El proceso constarán de varias fases y consistirá en una comunicación entre los agentes y el servidor de configuración. Estas fases son las siguientes:

En el servidor de configuración:

1. Recibirá una orden de inserción o borrado de algún usuario vía una API REST.

2. Publicará en el canal Pub-Sub "migration:new" el mensaje "NEW", para notificar a todos los agentes que se va a realizar una migración.

3. Cada vez que reciba un mensaje del canal "agent:OK" de un agente marcado como activo, marcará este como listo para la sincronización.

4. Comprobará cada cuatro segundos el número de Agentes activos preparados para la sincronización.

5. En el caso de que todos los agentes estén listos para la sincronización, realizará el cálculo del anillo de hashing y realizará la redistribución de claves pertinente.

6. Almacenará una copia del anillo de hashing y de la información de los nodos en la instancia Redis de comunicación.

7. Notificará a todos los agentes de la nueva migración mediante la publicación del mensaje "OK" en el canal "migration:new"

En cada uno de los agentes:

1. Publicará un saludo para notificar su presencia a todos los servidores de configuración mediante el canal "agent:new".

2. Comprobará si existe una migración en curso mediante la clave de control "MIGRATING". En caso afirmativo pasará al paso 4.

3. Esperará la notificación de una nueva migración mediante el mensaje "NEW" en el "canal migration:new".

4. Esperará la terminación de todas las peticiones entrantes y bloqueará nuevas peticiones. En el caso de que una petición tarde más de veinte segundos en completarse, abortará dicha petición.

5. Notificará su disponibilidad para la sincronización mediante un mensaje en el canal “agent:OK”.

6. Esperará la notificación de una migración satisfactoria y la posibilidad de migración mediante el mensaje “OK” en el “canal migration:new”.

7. Descargará de la instancia Redis de comunicación la nueva configuración.

8. Desbloqueará las peticiones entrantes y volverá a su modo normal de funcionamiento.

De está manera se asegura que todos los agentes disponen de la misma configuración. En caso de que exista algún problema en la migración desde alguno de los servidores de configuración, no se modificará el anillo de hashing y todos los agentes mantendrán la configuración actual.

La sincronización entre los servidores de comunicación es más simple. Cuando uno nuevo entra a formar parte del sistema, obtiene una copia de la configuración actual. De esta manera una migración en cualquiera de los servidores tendrá el mismo resultado sobre los datos, se mantendrá la consistencia.

Para evitar que dos servidores de comunicación modifiquen los mismos datos y puedan existir problemas en la sincronización, solo se podrá realizar una petición de migración a la vez. Para ello, cuando un servidor inicie un proceso de migración, notificará a los demás, que bloquearán todas las peticiones entrantes.

5.3. Tercera aproximación

La tercera aproximación diseñada modifica sobre todo el funcionamiento del algoritmo de distribución y redistribución. Se ha seguido un diseño basado en la “tercera estrategia” en Dynamo[Giu07], basada en la distribución fija de nodos virtuales4.1.2. En entornos de producción, la anterior estrategia seguida tenía muchos problemas de rendimiento y el bootstrapping era muy costoso.

5.3.1. Distribución

En esta aproximación la inserción de cada nodo no sigue un esquema aleatorio de distribución, sino que existen espacios o *slots* fijados de antemano por el sistema. Para ello se ha de dividir el anillo de hashing en partes iguales, que corresponderán a cada *slot*. Cada nodo virtual dispondrá de un espació de claves de tamaño fijo e igual que los demás nodos virtuales.

Creación del anillo de hashing.

Como se ha visto en las anteriores aproximaciones, un anillo de hashing es un array de valores ordenados y cuyos extremos están conectados. Es decir, el valor superior más próximo al último elemento del array es el primer elemento del array. En la distribución fija de nodos virtuales, este array tiene un tamaño fijo y prefijado desde su creación. En un anillo de hashing el espacio total de claves corresponde al rango de la función hash escogida, en el caso que nos ocupa, MD5. La construcción del anillo de hashing sigue el siguiente algoritmo.

Algorithm 5.5 Creación del anillo de hashing en una estrategia de distribución fija de nodos virtuales.

Require: $keySpace$= rango del espacio de claves de la función hash. $nPartitions$ = número de divisiones del anillo. $keyRing$ = anillo de hashing

Require: $nPartitions = 2^n, n \in \mathbb{N}$ and $nPartitions <= keySpace$

1: $chunkLength = keySpace/nPartitions$
2: **for** $i = 0$ to $nPartitions$ **do**
3: $keyRing$.push($i * chunkLength$)
4: **end for**

De esta manera queda creado el anillo de hashing. En el momento de su creación no existe ninguna asociación de un nodo virtual con un espacio de claves; esta asociación se creará en el momento de insertar un nuevo nodo Redis en el sistema.

Inserción de un nuevo nodo

A la hora de insertar un nuevo nodo, cada uno de los nodos virtuales de este "robará" *slots* a los demás nodos virtuales. Después de la inserción de un nuevo nodo, el número de nodos virtuales en el sistema debe ser proporcional al peso del nodo padre en el sistema, es decir, su capacidad de almacenamiento. Cuando se inserta un nuevo nodo, el sistema intenta sustituir los nodos virtuales del sistema por aquellos del nodo a insertar de la manera más uniforme posible. Además, la distribución a lo largo de todo el anillo también debe ser uniforme.

Para asegurar esta uniformidad tanto en el número de nodos virtuales como en la distribución, la cantidad de nodos virtuales que se insertan en el anillo de hashing se calcula de la siguiente manera: $S = \llcorner Q/N \lrcorner$, donde Q es el número total de slots, y N el número de nodos en el sistema. Cuando se inserte un nuevo nodo, el algoritmo de inserción irá recorriendo el anillo en sentido horario, empezando por la posición $N - 1$ (empezando el array en la posición 0), y realizando saltos de N espacios de claves. Esto asegura que el sistema contenga un número aproximadamente igual de nodos virtuales en el anillo de hashing.

Algorithm 5.6 Distribución de nodos virtuales en la inserción de un nodo

Require: $nPartitions$ = número de divisiones del anillo. $keyRing$ = anillo de hashing, $numNodes$ número de nodos en el sistema. $nodeName$ = nombre del nuevo nodo

Require: $nPartitions = 2^n, n \in \mathbb{N}$ and $nPartitions \leq keySpace$

1: $replicas = nPartitions/numNodes$
2: $i = numNodes - 1$
3: **for** $j = 1$ to $j = replicas$ **do**
4: CREATEASOC(keyRing[i], nodeName) //asocia el espacio de claves con el nodo (nodo virtual)
5: $i = i + numNodes$
6: **end for**

En el caso de no existir una división exacta entre el número de espacios de claves y el número de nodos, la distribución no será completamente uniforme. Esto no es ningún problema cuando $Q \gg N$, ya que la relación $\frac{Q/N}{Q/N+1} \approx 1$. Por lo tanto, es responsabilidad del usuario determinar un número de particiones Q lo suficientemente alto como para que la distribución de nodos virtuales sea uniforme.

Eliminación de un nodo

La eliminación de un nodo conlleva la cesión de los slots que ocupaba el nodo a eliminar. Los slots cedidos tienen que ser repartidos de manera uniforme entre los nodos del sistema para que se cumpla la propiedad $S_1 = S_2 = S_3 \cdots$. El algoritmo recorrerá el anillo de hashing en sentido horario, buscando los slots ocupados por los nodos virtuales a eliminar. Cada vez que se encuentre un nuevo slot a borrar, este se asignará a uno de los nodos del sistema, uno cada vez. El algoritmo es el siguiente:

Algorithm 5.7 Distribución de nodos virtuales en la eliminación de un nodo

Require: $nPartitions$ = número de divisiones del anillo. $keyRing$ = anillo de hashing, $numNodes$ número de nodos en el sistema. $nodeName$ = nombre del nuevo nodo

Require: $nPartitions = 2^n, n \in \mathbb{N}$ and $nPartitions \leq keySpace$

1: $replicas = nPartitions/numNodes$
2: $i = numNodes - 1$
3: **for** $j = 1$ to $j = replicas$ **do**
4: CREATEASOC(keyRing[i], nodeName) //asocia el espacio de claves con el nodo (nodo virtual)
5: $i = i + numNodes$
6: **end for**

Etiquetado de claves

Dada una configuración fija del anillo de hashing, el tamaño total del espacio de claves y el número de particiones, una clave concreta siempre pertenecerá al mismo slot. Ya que los tamaños y rangos de los slots es fijo, una clave siempre pertenecerá a un slot concreto. El hash de una clave siempre pertenecerá al mismo espacio de claves.

Esta propiedad es aprovechada por el sistema para redistribuir las claves en caso de eliminación o inserción de un nodo. El etiquetado de claves permite calcular a que slot pertenece una clave una sola vez, mejorando el rendimiento significativamente. El proceso de etiquetado es el siguiente.

1. El sistema calcula el hash de la clave mediante el algoritmo de hashing configurado previamente en el sistema (MD5, SHA...).

2. El algoritmo decide a que slot del anillo de hashing pertenece esta clave, realizando una búsqueda dicotómica del hash de la clave.

3. La clave se etiqueta añadiéndole un sufijo de la forma "clave{*slot*}", donde *slot* es el espacio de claves al que pertenece dicha clave.

4. El sistema decide el nodo en el cual se almacenará la clave. Cada slot está asociado con un nodo virtual.

5. La clave se almacena en el nodo requerido.

Esta aproximación permite buscar todas las claves pertenecientes a un cierto espacio de claves utilizando una expresión regular. Por lo tanto, no es necesario extraer todas las claves de un determinado nodo e ir analizando su pertenencia o no a cierto espacio de claves.

5.3.2. Redistribución

Para la redistribución de nodos se siguió una aproximación más eficiente que las dos anteriores. Como se ha visto en la sección anterior, todas las claves introducidas se modifican añadiendo un sufijo que indica a que slot del anillo de hashing pertenecen. El algoritmo de redistribución que se utiliza en este caso es mucho más eficiente gracias al uso de este etiquetado, que permite obtener todas las claves pertenecientes a cierto slot y no tener que calcularlas una a una. El sistema analizará que espacios de claves han actualizado su pertenencia respecto a los nodos virtuales y migrar las claves pertinentes al nuevo nodo.

Inserción de un nodo

Cuando un nuevo nodo es añadido al sistema, sus nodos virtuales serán emplazados en espacios de claves ya existentes. Estos nodos virtuales sustituirán aquellos ya pertenecientes al anillo de hashing de una manera uniforme. La situación ideal del anillo de hashing después de esta operación es aquella en la que existe la misma cantidad de nodos virtuales de cada instancia.

Las claves contenidas en los espacios de claves modificados deberán migrar al nuevo nodo insertado. Los nodos virtuales que anteriormente poseían estos espacios de claves son los que contendrán las claves a migrar. Está información, los nodos que contienen las claves a migrar, se obtiene mediante el algoritmo de inserción en el anillo de hashing. Cuando se calcula la distribución de nodos en la inserción de una nueva instancia redis, se obtiene un objeto que sigue el siguiente esquema:

$$\begin{cases} nodo_1 : & [slot_1 \cdots slot_n] \\ \vdots & \vdots \\ nodo_n & [slot_1 \cdots slot_n] \end{cases}$$

$\{nodo_1 \cdots nodo_n\}$ es el conjunto de nodos que poseían algún nodo virtual sobrescrito en la inserción. $[slot_1 \cdots slot_n]$es un vector que contiene los slots donde se encontraban dichos nodos virtuales.

El algoritmo seguido para el migrado es el siguiente.

Algorithm 5.8 Redistribución de claves en la inserción de un nodo

Require: *oldNodes* = objeto que contiene los nodos virtuales que han sido sobreescritos
Require: *newNode* = nodo a añadir al sistema
1: **for** *node* in *oldNodes* **do**
2: **for** $i = 1$ to $i = node.keys.length$ **do**
3: *keys* = GETKEYS(node.keys[i]);
4: MIGRATE(newNode, node, keys); // node –> newNode
5: **end for**
6: **end for**

La obtención de todas las claves pertenecientes a un slot dentro del anillo de hashing y almacenadas en una instancia Redis se obtiene mediante el comando Redis "KEYS pattern", donde "pattern" sería una expresión regular. En este caso la expresión regular será "*{*slot*}", donde slot es el la etiqueta del espacio de claves concreto. Este comando devolverá un array de claves pertenecientes a este slot, y serán las claves a migrar.

Eliminación de un nodo

Cuando un nodo abandona el sistema, los slots ocupados por los nodos virtuales de este se distribuirán de forma equitativa entre los demás nodos, de manera que

el anillo de hashing contenga el mismo número de nodos virtuales de cada base de datos.

Los las claves almacenadas en el nodo a eliminar deberán ser redistribuidas entre los demás nodos del sistema. Las claves se distribuirán de manera uniforme entre los nodos restantes, atendiendo al nuevo nodo virtual que ocupaba el slot abandonado. El funcionamiento es el mismo que en el caso de la inserción. La función de distribución en caso de eliminación devuelve un objeto que contiene la asociación entre slots y nodos en la nueva distribución.

$$\begin{cases} nodo_1: & [slot_1 \cdots slot_n] \\ \vdots & \vdots \\ nodo_n & [slot_1 \cdots slot_n] \end{cases}$$

$\{nodo_1 \cdots nodo_n\}$ es el conjunto de nodos que han sobrescrito al nodo eliminado. $[slot_1 \cdots slot_n]$ es un vector que contiene los slots que contenían los nodos virtuales eliminados.

Algorithm 5.9 Redistribución de claves en la eliminación de un nodo

Require: *newNodes* = objeto que contiene la asociación de nodos y slots en la nueva distribución
Require: *delNode* = nodo a eliminar sistema
1: **for** *node* in *newNodes* **do**
2: **for** $i = 1$ to $i = node.keys.length$ **do**
3: *keys* = GETKEYS(node.keys[i]);
4: MIGRATE(node, delNode, keys);// delNode –> node
5: **end for**
6: **end for**

5.3.3. Alta disponibilidad

En esta tercera aproximación se contemplan todos los casos de fallo en cada nodo del sistema ya sean agentes, servidores de configuración, bases de datos de persistencia o bases de datos de configuración. Cada elemento es replicable dentro del sistema de tal manera que no existan puntos únicos de fallo y que redundancia de datos esté asegurada.

La comunicación y detección de fallos se lleva a cabo mediante el sistema Redis y Redis Sentinel. Toda comunicación entre los servidores y los agentes se realiza mediante canales Pub/Sub, y la detección de fallos y lanzamiento de failovers se llevará a cabo mediante monitores o "sentinels" del sistema.

Arquitectura

La arquitectura consiste en un conjunto de monitores (sentinels) que monitorizan el estado de las bases de datos en informan de posibles problemas. Estos monitores se comunican con los nodos de configuración que son los encargados de replicar la información de estado en todo el sistema. Existen dos tipos de clústers de base de datos: los clústers de persistencia y el clúster de configuración. Los primeros pueden ser varios y almacenan la información persistente de PopBox (colas y transacciones). Su número depende de la carga del sistema. El segundo es el medio de comunicación para la sincronización de la información de distribución de carga. Comunicará a los agentes la modificación del anillo de hashing y la actualizará.

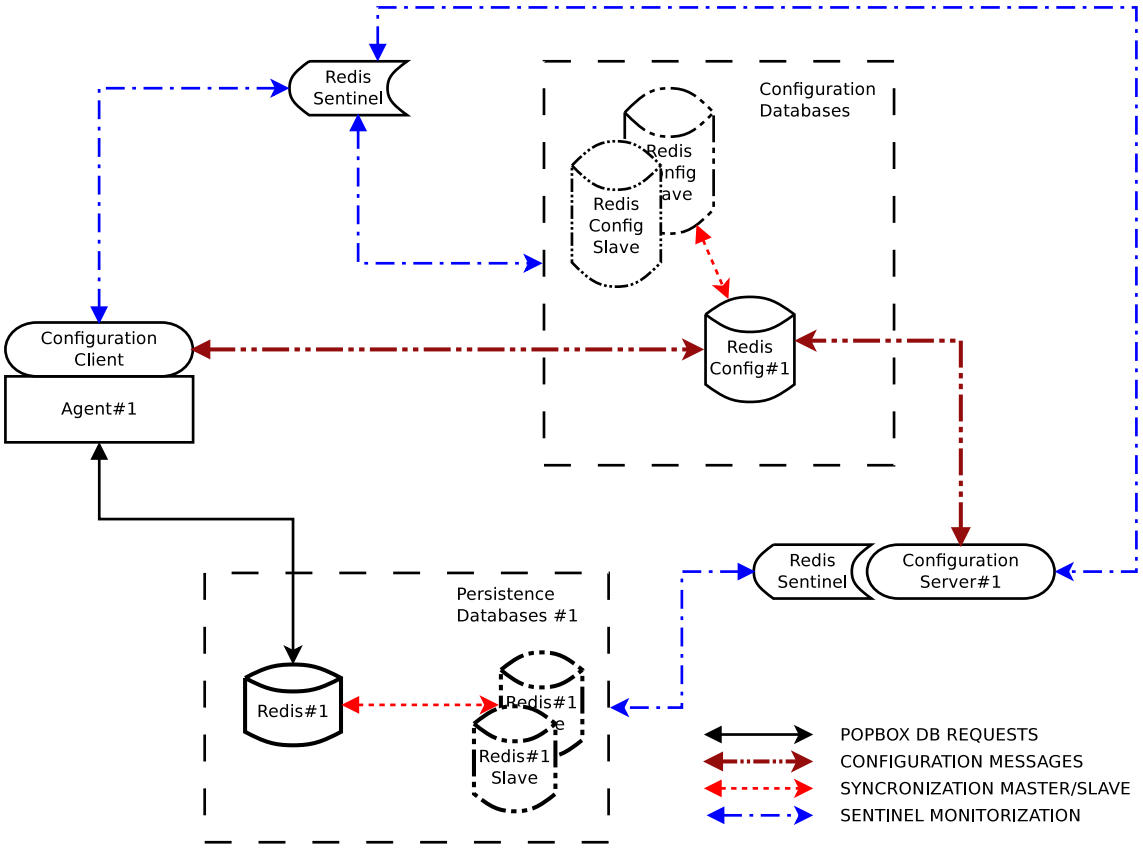

Figura 5.3.1: Arquitectura de alta disponibilidad en la tercera aproximación

En esta vista ningún elemento del sistema está duplicado por simplificar la vista, pero todos los elementos son replicables.

Todas las comunicaciones se realizan mediante el protocolo Redis.

Existen dos actores principales en la gestión de la alta disponibilidad del sistema, que son los servidor de configuración y los clientes de configuración. Además de estos dos, existen otros actores, los Sentinels, que son transversales a todo el sistema y que se encargan de monitorizar las instancias Redis y realizar cambios sobre ellas.

Servidores de configuración

Los servidores de configuración se encargan de calcular la distribución de los nodos en el anillo de hashing, la redistribución de claves y la sincronización con todos los agentes. Respecto a la alta disponibilidad, los servidores de configuración monitorizarán cada uno de las instancias Redis mediante agentes Sentinel, que son transversales al sistema.

Pueden existir varios servidores de configuración y cada uno tendrá una interfaz accesible vía HTTP para insertar o eliminar nodos del sistema.

Cada servidor de configuración creará una nueva instancia de Sentinel, en un puerto libre, que se encargará de monitorizar todas las bases de datos de persistencia existentes en el sistema. En el caso de añadir o eliminar una instancia Redis de la base de datos, el monitor Sentinel se reiniciará con una nueva configuración, monitorizando de nuevo a todos los nodos del sistema.

Así mismo, un servidor de configuración se conectará a un Sentinel definido por defecto que será el monitor de la base de datos de configuración. Esta descubrimiento del Sentinel no es automático y requiere ser conocido tanto por el servidor de configuración como por el cliente de configuración. La razón de esta configuración no dinámica se verán más adelante.

Cada servidor de configuración, mediante comunicación con su Sentinel, puede detectar si hay algún servidor maestro de alguno de los clústers fallando. Cuando esto sucede se llevan a cabo las siguientes acciones:

1. Cada Sentinel asociado a un servidor de configuración detecta el fallo y lo notifica mediante Pub/Sub a los demás Sentinels.

2. Cuando se ha alcanza el quórum requerido (el número de Sentinels que marquen la instancia Redis como fallando), se inicia un proceso de failover.

3. Al terminar este proceso, se promociona a alguno de los esclavos del nodo que ha fallado.

4. Cuando el servidor de configuración asociado al Sentinel que ha iniciado el proceso de failover recibe notificación de la finalización de este, comienza un proceso de sincronización.

5. El servidor de configuración espera que todos los agentes estén listos para una sincronización.

6. El servidor de configuración modifica la configuración de los nodos en la base de datos de configuración.

7. Los agentes reciben la nueva configuración (cambio de maestro) y siguen operando.

Clientes de configuración

Los clientes de configuración son los encargados de realizar las tareas de sincronización en el lado del Agente. En el caso de la alta disponibilidad, se encargarán de controlar que la base de datos de configuración no falle.

Cada cliente de configuración está comunicado con un Sentinel que monitoriza el estado de la base de datos de configuración. Cada cliente puede estar conectado a un único Sentinel, pero un Sentinel puede estar comunicado con varios clientes. Como mínimo debe existir un Sentinel en el sistema para que se pueda controlar el estado de la base de datos, que será además un canal de comunicación

En caso de un fallo en alguna instancia Redis de configuración, el cliente de configuración direccionará las comunicaciones con el cliente hacia el nuevo Redis maestro dentro del clúster. Así mismo, los nuevos Agentes que se incorporen al sistema obtendrán la configuración de la base de datos maestra que indique el Sentinel al que estén conectados, además de realizar la comunicación con el servidor de configuración a través de esta.

El descubrimiento de la base de datos de configuración mediante un monitor Sentinel se realiza en 3 pasos:

1. Comunicación de estado entre el Sentinel y la base de datos de configuración para obtener la instancia maestra.
2. Comunicación mediante protocolo Redis entre el cliente de comunicación y el Sentinel. El Sentinel informará al cliente de cual es el nodo maestro del sistema de configuración.
3. Conexión del cliente al nodo maestro.

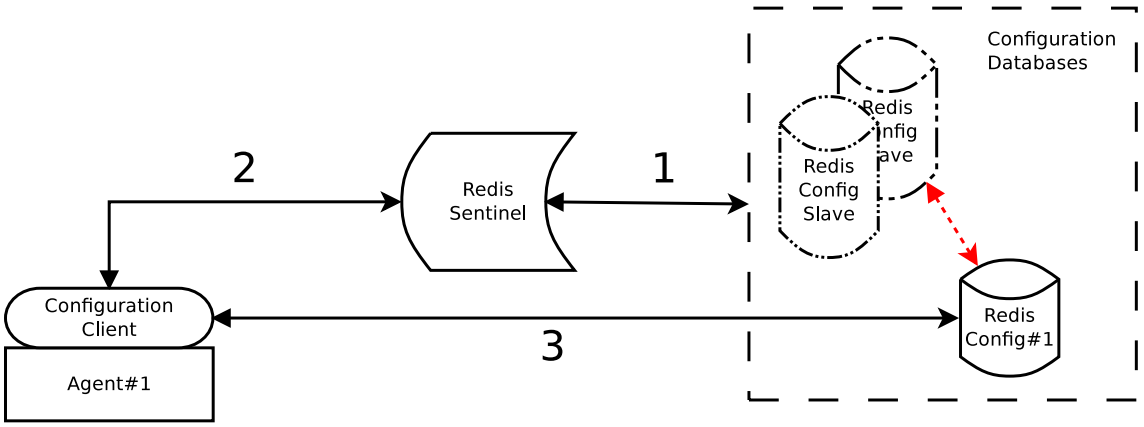

Figura 5.3.2: Descubrimiento de la base de datos de configuración mediante un Sentinel

Capítulo 6

Conclusiones

La importancia actual de los servicios en la nube plantea un reto a las soluciones de alta disponibilidad y escalabilidad existentes. Cada vez se necesitan servidores más flexibles y más escalables, que sean capaces de atender a cambios bruscos en la demanda del servicio. El Departamento de Desarrollo Transversal de Telefónica I+D es consciente de esto y requiere de soluciones HA y de escalabilidad para sus servicios PopBox y Rush.

En este trabajo se han abordado diferentes soluciones desde el punto de vista técnico. Se ha aplicado una combinación lo más eficiente posible de las soluciones aplicadas actualmente por empresas como Amazon, Last.fm o LinkedIn para adaptarse a los requerimientos del sistema PopBox. Durante el desarrollo del trabajo se realizaron varias aproximaciones, evaluando cada solución de una manera crítica.

Este proyecto cubre la necesidad de una solución de escalabilidad, elasticidad y alta disponibilidad en la base de datos Redis. Otras bases de datos NoSQL como Cassandra, MongoDB o Riak proveen estas características. La solución implementada es apta para ser una aproximación genérica a este problema. Este trabajo propone alguna soluciones no implementadas aún en esta base de datos.

Personalmente, este proyecto ha ampliado enormemente mis conocimientos sobre el mundo de la computación en la nube. Las grandes empresas están realizando muchos avances en todo lo relacionado con este campo y he tenido la oportunidad de analizar el trabajo realizado por estas. Particularmente, la documentación sobre el trabajo realizado por los ingenieros de Amazon en su base de datos DynamoDB me ha sido de gran ayuda y ha ampliado mi visión sobre las estrategias de escalabilidad.

Además he podido tener un profundo contacto con NodeJS, una tecnología que, en mi opinión, va a ser un punto de inflexión en el mercado de las aplicaciones cloud. Durante la realización de este proyecto me he enfrentado a problemas reales de eficiencia y optimización, y he diseñado soluciones utilizando la potencia de este sistema.

Al ser NodeJS un servidor Javascript, he llegado a conocer este lenguaje en profundidad. He aprendido a manejar el paradigma de la programación orientada a

eventos y la asincronía de este lenguaje, que cada vez tiene más cabida en el ámbito empresarial.

He profundizado mucho en la tecnología Redis, una base de datos usada por empresas como Twitter y que está teniendo muchísimo impacto en el mundo del software libre y las startups. He podido desarrollar un sistema completo usando esta tecnología, haciendo uso del sistema Pub/Sub, realizando optimizaciones y creando un sistema de sharding apoyado en este.

Finalmente, trabajar en el entorno profesional de una gran empresa de innovación como es Telefónica Digital me ha proporcionado una visión del trabajo distinta a la del ámbito académico. He aprendido a trabajar en un entorno colaborativo a gran escala y a realizar informes detallando mi implicación en el trabajo, además de adaptarme rápidamente a los cambios internos.

Capítulo 7

Líneas futuras

Durante los próximos meses, el sistema de alta disponibilidad y escalabilidad de PopBox seguirá su desarrollo y mejora dentro del marco de trabajo del Laboratorio de Innovación Abierta de la Facultad de informática.

Este plan de mejoras incluirá los siguientes aspectos, en orden de prioridad:

1. Mayor eficiencia en el resharding. Actualmente la redistribución de claves utiliza la búsqueda de claves por patrón, algo desaconsejado para sistemas en producción. En la tercera aproximación del desarrollo5.3.2 se realiza una mejora parcial, buscando mediante patrón solo las claves involucradas en la redistribución. El futuro de esta implementación es guardar cada espacio de claves en un SET Redis, ya que su tiempo de accesibilidad es mucho menor. En esta línea se tendrá que lidiar con el problema de las claves expiradas dentro de un SET, ya que Redis no proporciona mecanismos para notificar las expiraciones dentro de un conjunto de claves.

2. Distribución de escrituras y lecturas. Las lecturas de datos se realizarán sobre los diferentes esclavos. Esto permitirá distribuir la carga de lecturas y disminuir la carga en el maestro. Actualmente tanto lecturas y escrituras se realizarán en el maestro, delegando toda la carga en este.

3. Distribución por etiquetas. Al igual que en otros sistemas, la distribución entre bases de datos atendiendo a la localización geográfica, el nombre de usuario, .etc, resulta en baja latencia en los accesos. El objetivo será poder distribuir las claves atendiendo a algún parámetro personalizable.

4. Notificaciones al administrador. Se implementará un sistema capaz de notificar al administrador el fallo de alguna base de datos o algún componente del sistema. Actualmente la información del estado del sistema se obtiene bajo demanda. Se creará algún componente que avise al administrador vía email o similar.

5. Mayor flexibilidad de configuración en el sistema HA. Actualmente sólo algunos parámetros del sistema HA son configurables por el usuario

Bibliografía

[Abe11] Michael Abernethy. Just what is Node.js? *IBM Developer Works*, page 10, 2011. 2.1

[Cod] Codefutures. Database sharding. *Codefutures*. 4.1.2

[Dav97] Eric Lehman Tom Leighton Mathhew Levine Daniel Lewin Rina Panigrahy David Karger. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. *ACM Symposium on Theory of Computing*, 1997. 4.1.2, 4.1.2, 4.1.2

[FC06] Sanjay Ghemawat Wilson C. Hsieh Deborah A. Wallach Mike Burrows Tushar Chandra Andrew Fikes Robert E. Gruber Fay Chang, Jeffrey Dean. Bigtable: A distributed storage system for structured data. *Google, Inc.*, 2006. 4.1.2

[Fie00] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, UNIVERSITY OF CALIFORNIA, IRVINE, 2000. 3.1

[Giu07] G. D. Giuseppe DeCandia et al. Dynamo: Amazon's Highly Available Key-value Store. *All Things Distributed*, 2007. 4.1.2, 5.3

[Inc11] Exforsys Inc. What is Monkey Testing. *Exforsys Inc.*, 2011. 5

[Jon07] Richard Jones. libketama - a consistent hashing algo for memcache clients. *Last.fm Journal*, 2007. 4.1.2

[Kan12] Chris Kanaracus. Gartner: SaaS market to grow 17.9 % to 14.5B. *Computer World*, 2012. 1.1

[Men12] Manuel Angel Mendez. El BBVA se pasa a Google Apps. *El Pais*, 2012. 1.1

[MS03] Evan Marcus and Hal Stern. Blueprints for high availability. 2003. 4.2, 4.2.4

[Nie09] Michael Nielsen. Consistent hashing. *michaelnielsen*, Junio 2009. 4.1.2

[Ove12] Sam Overton. Virtual Nodes Strategies. *Acunu*, 2012. 4.1.2

[San] Salvatore Sanfilippo. KEYS pattern. 5.1.5

[San13] Salvatore Sanfilippo. News about Redis: 2.8 is shaping, I'm back on Cluster. *Antirez Blog*, 2013. 2.2.5

Este documento esta firmado por

	**Firmante**	CN=tfgm.fi.upm.es, OU=CCFI, O=Facultad de Informatica - UPM, C=ES
	Fecha/Hora	Fri Feb 14 19:51:42 CET 2014
	Emisor del Certificado	EMAILADDRESS=camanager@fi.upm.es, CN=CA Facultad de Informatica, O=Facultad de Informatica - UPM, C=ES
	Numero de Serie	630
	Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)